Efficient Implementation of OpenACC cache Directive on NVIDIA GPUs

Ahmad Lashgar

ECE Department, University of Victoria
Address: Room A220, Engineering Lab Wing Building,
University of Victoria, Victoria, BC, Canada
Email: lashgar@uvic.ca

Amirali Baniasadi

ECE Department, University of Victoria
Address: Room 323, Engineering Office Wing Building,
University of Victoria, Victoria, BC, Canada
Email: amiralib@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. The *cache* directive is among the challenging to implement directives. 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 undesirably 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. 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 2.4X (compared to the baseline OpenACC.)

Keywords: OpenACC; Cache memory; CUDA; Software-managed cache; Performance;

Reference to this paper should be made as follows: Lashgar, A. and Baniasadi, A. (2017) 'Efficient Implementation of OpenACC cache Directive on NVIDIA GPUs', *Int. Journal of High Performance Computing and Networks*, Vol. x, No. x, pp. xxx–xxx.

Biographical notes: Ahmad Lashgar is a fourth year PhD Candidate in Electrical and Computer Engineering at University of Victoria, BC, Canada. For completing his PhD, he works on developing hardware/software optimizations for accelerators. He wrote IPMACC compiler for translating OpenACC applications to CUDA, OpenCL, and ISPC backends.

Amirali Baniasadi received his PhD degree in computer engineering from Northwestern University, Evanston, IL, USA in 2002. He is currently professor at the ECE department of University of Victoria, Victoria, BC. His current research interests include high-level accelerator programming models, low-power microarchitecture, complexity-effective design and clustered processors.

1 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 (while we focus on CUDA in this paper, most of the discussions apply to OpenCL as well.). 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 [Herdman2012]. 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 [Lashgar2015] 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 paper investigating the implementation aspect of the *cache* directive. We show that a naïve implementation hardly improves performance (presented in Section 2). We provide better understanding regarding implementation challenges and list compile-time optimizations and opportunities to enhance performance (presented in Section 4).

- We introduce three methods for implementing the *cache* directive (presented in Section 3). 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 5.1). 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 benchmarks (presented in Section 5.2): 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 2.4X, and almost match that of CUDA (while reducing the development effort by 24%).

The rest of this paper is organized as follows. In Section 2 we overview related background and discuss inefficiencies of a naïve cache implementation. In Section 3 we present our proposed implementations for the *cache* directive. In Section 4 we introduce optimizations applicable to the proposed implementations. In Section 5 we evaluate performance of the proposed methods. In Section 6 we discuss the limitations of our approach. In Section 7 we overview related work. Finally, in Section 8 we offer concluding remarks.

2 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 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)

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.

According to OpenACC specification [OpenACC2015], variables and subarrays listed in *var-list* should be fetched into the highest level of the cache for the body of the loop. We refer to the scope of the loop as *cache region*. In the *cache region*, all accesses to the variables and subarrays listed in *var-list* should be served from the cache.

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.

Listing 1: The cache directive example; 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 }
```

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.

3 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 (simplified versions of the presented methods are applicable for scalar variables.). 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 subarray range (we assume pointer aliasing is not the case and pointers are declared as restricted type in the accelerator region, using C's *restrict* keyword.). Using the information provided through the directive,

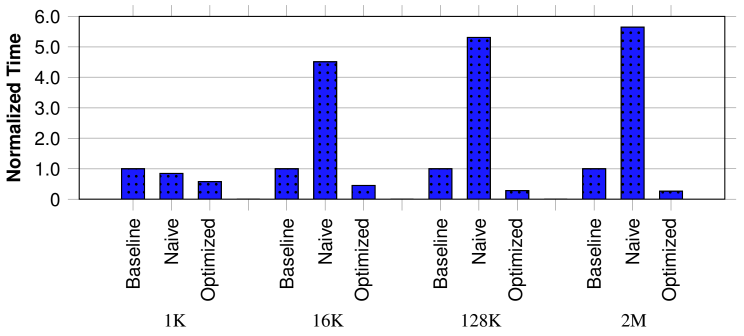

Figure 1: Comparing naïve and optimized cache implementations under 1D stencil kernel listed in Listing 1 (30-element radius, 1K, 16K, 128K, and 2M elements.)

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. As we show in this paper, this can be very expensive to calculate in runtime. Starting from OpenACC 2.5 [OpenACC2015], the following restriction has been added to the cache directive specification: within the cache region, all references to an array listed in the cache directive must refer to the range specified in the cache directive. Our first two proposed methods (EHC and RBC) comply with the older OpenACC specification [OpenACC2013] and are suitable for applications written in older OpenACC versions (e.g. v2.0). Our third method takes advantage of the restriction added in OpenACC 2.5 to highly optimize the implementation.

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.

3.1 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 a 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).

3.2 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 consecutive 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. plus a branch for 1D, $4 + 3$ logical ops. plus a branch for 2D, etc.).

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

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 [OpenACC2015]. This essentially means RBI can be used with all applications that follow OpenACC ≥ 2.5.

3.4 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()* (as a performance issue, these procedures are declared inline to avoid procedure calls within the accelerator region.). The accelerator code is generated to call *__cache_fetch()* 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 (alternative is discussed in Section 4.3.).

Listing 2: Implementation of Emulating Hardware Cache (EHC) in CUDA.

```
__device__ void __cache_fetch(PTRTYPE* g_ptr, PTRTYPE* c_ptr, unsigned* ctag_ptr,
  unsigned st_idx, unsigned en_idx){
    for(unsigned i=st_idx; i<en_idx; i++){
        unsigned cache_idx=acc_idx&0x0ff; //direct map
        c_ptr[cache_idx]=g_ptr[i];      // update data array
        ctag_ptr[cache_idx]=i;          // update tag array
    }
}
__device__ PTRTYPE __cache_read(PTRTYPE* g_ptr, PTRTYPE* c_ptr, unsigned* ctag_ptr,
  unsigned st_idx, unsigned en_idx, unsigned acc_idx){
    unsigned cache_idx=acc_idx&0x0ff; //direct map
    if(ctag_ptr[cache_idx]==acc_idx){
        return c_ptr[cache_idx];        // read from cache
    }else{
        c_ptr[cache_idx]=g_ptr[acc_idx]; // read from global memory, update data
        ctag_ptr[cache_idx]=acc_idx;      //    and tag arrays
        return c_ptr[cache_idx];          // read from cache
    }
}
__device__ void __cache_write(PTRTYPE* g_ptr, PTRTYPE* c_ptr, unsigned* ctag_ptr,
  unsigned st_idx, unsigned en_idx, unsigned acc_idx, PTRTYPE value){
    unsigned cache_idx=acc_idx&0x0ff; //direct map
    if(ctag_ptr[cache_idx]!=acc_idx)
        ctag_ptr[cache_idx]=acc_idx;           // update tag
    g_ptr[acc_idx] =c_ptr[cache_idx] =value; // write-through
}
```

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 the condition of the control statement is always true in RBI. In this Listing, the mapping is an arithmetic operation; subtracting index from the start pointer. *__cache_fetch()* routine in all implementations has a *for* loop statement. Later in Section 4.2.4, we discuss opportunities to accelerate this loop through parallelization.

4 Implementation Optimizations

In this section, we introduce optimizations for implementations introduced in the previous section. Specifically, we present optimizations for cache fetch routine, cache sharing, cache writes, and minimizing index mapping overhead.

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;
}
```

4.1 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 (the fetch routine might be called multiple times, if located in a sequential loop.). 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 3.4) 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, e.g. 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 4.2.

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
}
```

4.2 Cache Sharing

Considering the relative nesting of the *cache* directive in respect to parallel loops, there are two types of parallel loops: *outer parallel loops* and *inner parallel loops*. Iterations of inner parallel loops already share the same data. In this section we introduce a method to find data sharing among the iterations of outer parallel loops. Listing 4 clarifies outer and inner loops in an example.

The *cache* directive is located within one (or more) *outer* parallel loop(s) and the cache space should be allocated once per parallel instance of outer parallel loop(s). 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 the same thread block (in CUDA), sharing data through *CUDA shared memory*.

We have different methods for cache sharing under EHC, RBC, and RBI. Under EHC, cache sharing can be achieved by sharing one single larger data and tag arrays among all iterations. The complexity is in efficiently managing consistency of data and tag arrays, considering parallel accesses to the cache may occur from different iterations. Currently, the only mechanism in CUDA to maintain the consistency is to update data and tag arrays atomically using atomic operations. Since this severely slows down the performance, we found cache sharing unpromising in EHC. Below we discuss cache sharing method under RBC and RBI.

We decompose the cache sharing problem under RBC and RBI to five subproblems: i) extract sharing, ii) find sharing width, iii) renew cache scope, iv) fetch collaboratively, and v) optimize cache size. Below we discuss each problem.

4.2.1 Extract Sharing

The problem is to map outer parallel loops (loops that are marked by the OpenACC loop directive as parallelizable) to thread hierarchies with the constraint of maximizing the subarray overlap among threads of the thread block. Listing 5 presents a compiler pass as a solution to this problem. The problem inputs are the *cache* directive (code block where pragma is injected and list of subarrays), outer parallel loops (loop handle, induction variable, and increment step), and the kernel code. The problem output is the mapping of loop iterations to CUDA thread block dimensions.

Listing 5: Compiler pass that extracts cache sharing opportunity and suggests a mapping to maximize the overlap among subarrays of consecutive iterations.

```
Inputs:
      cache: the code block id of the cache region
  subarrays: array of subarrays listed in the cache directive
         Ls: array of outer parallel loops, indexed by induction variables
        IDs: array of induction variables associated with outer parallel loops
       code: the kernel code
Output:
    mapping: structure showing the parallel loops to kernel dimensions mapping
Begin
  final_mapping = []
  skipped_subarray = []
  for subarray in subarrays
    unmapped_dimensions = [x, y, z]
    suba_mapping = []
    for dimension in subarray
      lower, length <— get_specifiers(dimension)
      if is_linear(lower, IDs, code, Ls)
        rate, inductionVar, offset <— get_linear_params(lower, IDs, code, Ls)
        // map parallel loop iterated by inductionVar to an unmapped dimension
        suba_mapping.push(Ls[inductionVar] —> unmapped_dimension.pop())
    if not is_contrary(final_mapping, suba_mapping)
      mapping = merge(final_mapping, suba_mapping)
    else
      skipped_subarrays.push(subarray)
  return final_mapping
End
```

The pass iterates over the subarrays listed in the *cache* directive. For each dimension of the subarray (dealing with multi-dimensional subarrays), *lower* and *length* specifiers are read. If *lower* is a linear function of one single induction variable, consecutive iterations of the loop corresponding to the induction variable are considered for sharing (see examples in Table 1). *is_linear()* function returns true if i) *lower* specifier is a linear function of an induction variable and ii) the increment step of tthe corresponding loop is linear (e.g. i+=1, i-=1, i+=7, etc.). If *lower* is linear, it should be in $rate * inductionVar + offset$ form, where *inductionVar* is an induction variable and *rate* and *offset* are expressions independent of any induction variable. Forcing the increment step to be linear assures that the neighbor threads cache subsequent elements, forming a sharing range that is densely populated by the data from neighbor threads (consecutive iterations).

get_linear_params() returns *rate*, *inductionVar*, and *offset*. *suba_mapping* is updated to map the parallel loop iterated by *inductionVar* to unmapped thread block dimensions, starting with *x* dimension. *is_contrary()* returns true if the *suba_mapping* that is found here contrasts with the mapping recorded in *final_mapping*. If this is the case, *subarray* is pushed to *skipped_subarrays*. Cache sharing optimizations will be skipped for the subarrays in *skipped_subarrays*. Otherwise, *final_mapping* is updated to be merged with *suba_mapping*.

Table 1 Example of cache sharing when *lower* specifier is a linear function of an induction variable. Assumptions: *i* is an induction variable of a parallel loop, increment step of the loop iterated by *i* is *+1*, and thread block size is 3.

subarray	lower	length	ranges mapped to the iterations	shared range
a[i:3]	i	3	T_0 -> 0 to 2 T_2 -> 1 to 3 T_2 -> 2 to 4 etc.	T_0 to T_2 -> 0 to 4 etc.
a[2*i+1:3]	2*i+1	3	T_0 -> 1 to 3 T_1 -> 3 to 5 T_2 -> 5 to 7 etc.	T_0 to T_2 -> 1 to 7 etc.
a[3*i+4:5]	3*i+4	5	T_0 -> 4 to 8 T_1 -> 7 to 11 T_2 -> 10 to 14 etc.	T_0 to T_2 -> 8 to 14 etc.

4.2.2 Find Sharing Width

Sharing width is referred to the number of iterations (or threads) that share one common cache. Ideally, sharing width is equal to the thread block size. This is the case when the total number of loop iterations is multiple of the thread block size. However, since the total number of loop iterations is a runtime variable mostly, compiler cannot statically assure this number is multiple of thread block size. We propose three different methods to find the sharing width in CUDA; using synchronization, kernel arguments, or fixed.

Synchronization: This method counts the number of threads that have reached the cache region. To count the number of threads, *__syncthreads_count(bool flag)* device function from CUDA API is used. To count the number of threads along *x* dimension of the thread block, *__syncthreads_count* is called with the argument *threadIdx.y==0 && threadIdx.z==0*. Similarly, for *y* and *z* dimensions of the thread block, the function is called with *threadIdx.x==0 && threadIdx.z==0* and *threadIdx.x==0 && threadIdx.y==0* arguments, respectively.

Kernel Arguments: This method exploits the fact that only the last thread blocks across every dimension may have a sharing width different than the thread block size. This width can be pre-calculated and passed to the kernel as an argument, knowing the total number of iterations and the thread block size upon kernel launch. Within the kernel, threads check if they belong to the last thread block of the dimension. If yes, sharing width is set to the value passed as the argument. Otherwise, sharing width is equal to the thread block size. This method has a performance advantage over the first method as it avoids synchronization and reduction.

Fixed: This method simply sets the sharing width equal to the thread block size. This method is only applicable in the case where compiler can statically assure that the total number of loop iterations is multiple of the thread block size.

4.2.3 Renew Cache Scope

From the notation of the *cache* directive, every thread knows the range from *lower* to *lower* + *length* is cached. For RBC and RBI, *start* and *end* pointers are set to these values. However, when threads of the thread block are sharing the cache, these pointers should be recalculated, since a larger data range is cached in this case. We propose two different methods to recalculate pointers: communicating and private.

Communicating: This method shares pointers among threads of the thread block. To share pointers, these are declared as CUDA *__shared__* variables. To set pointers consistently, one thread is to set *start* and another thread is to set *end*. *start* pointer is set to *lower* by the thread that is demanding subarray's elements located at the lowest address. This is the first thread within the sharing width, if the corresponding loop has increasing increment step (e.g. +=1, +=3, etc.). Otherwise, if the corresponding loop has decreasing increment step (e.g. -=1, -=3, etc.), this thread is the last thread within the sharing width. Similarly, *end* pointer is set to *lower* + *length* by the thread that is demanding subarray's elements located at the highest address. This is the last thread within the sharing width, if the corresponding loop has increasing increment step. Otherwise, if the corresponding loop has decreasing increment step, this thread is the first thread within the sharing width.

Private: This method allocates *start* and *end* pointers privately for each thread. Following equations are used to recalculate *start* and *end* pointers privately:

$start = lower - rate * threadID$

$end = start + (length - 1) + rate * (sharingWidth - 1)$

where *lower*, *rate*, *threadID*, *length*, and *sharingWidth* parameters are explained below. *lower* and *length* are specifiers of the subarray passed to the *cache* directive. *rate* is obtained from *lower* by using *get_linear_params()* function explained in Section 4.2.1. *sharingWidth* is the number of active threads in the cache region obtained by the methods discussed in Section 4.2.2. *threadID* is the thread ID within the thread block, ranging from 0 to *sharingWidth - 1* in the cache region. Equations above are applicable to the case where *lower* is a function of an induction variable of a loop with an increasing increment step. Under decreasing increment step, following equations are used:

$start = lower + rate * threadID - rate * (sharingWidth - 1)$

$end = lower + rate * threadID + (length - 1)$

4.2.4 Fetch Collaboratively

If cache sharing is applicable, threads of the thread block share one common data in shared memory. Since the common data is composed of words located at consecutive addresses, threads of the thread block can be used to efficiently fetch the data using few well-coalesced accesses in parallel. To perform this optimization, only *__cache_fetch()* routine in Listing 3 needs to be modified. The *for* loop statement should be modified to:

for(unsigned i=threadIdx.x+st_idx; i<en_idx; i+=blockDim.x)

This is for the case where the subarray is one-dimensional and the parallel loop is mapped to *x* dimension of the thread block. For multidimensional subarrays, this loop is replicated but modified to reflect correct mapping of parallel loops to thread block dimensions.

4.2.5 Optimize Cache Size

When cache is not shared, each thread demands *length* elements from shared memory. While sharing the cache among threads of the thread block, it might seems $length *$

$sharingWidth$ elements from shared memory are required. This is correct as long as subarrays of consecutive loop iterations are located back to back in the memory. Otherwise, if there is an overlap or gap among subarrays, this number overestimates or underestimates the exact size. We use the following formula to optimize the cache size:

$length + rate * (sharingWidth - 1)$

where *sharingWidth* is the number of active threads in the cache region obtained by the methods discussed in Section 4.2.2. *length* is a specifier of subarray passed in to the *cache* directive. *rate* is obtained from *lower* by using *get_linear_params()* function explained in Section 4.2.1.

4.3 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 5.3.

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.

4.4 Index Mapping

As we discussed in Section 3, 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.

5 Experimental Results

In this section, we first report the experiments performed to understand shared memory and optimize our implementation on the target GPU. Then we study the performance of methods introduced in Section 3, under three test cases. This is followed by investigating performance of different cache write policies. Finally, we evaluate performance portability of our implementation.

Listing 6: 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];
}
```

We use IPMACC compiler [Lashgar2014] for compiling OpenACC applications and implementing the *cache* directive. IPMACC framework translates OpenACC to CUDA and uses NVIDIA *nvcc* compiler to generate GPU binaries. We run evaluations under NVIDIA Tesla K20c GPU. The execution time of the kernel is measured by nvprof [NVIDIA2017a]. Every number is harmonic mean of 30 independent samples.

5.1 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 warp (group of threads executing instructions in lock-step over the SIMD). 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

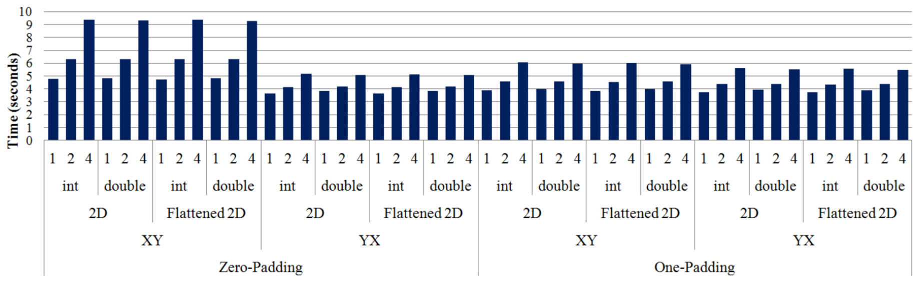

Figure 2: Comparing execution time of kernel under various shared memory configurations.

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.

5.1.1 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 6. 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.

5.1.2 Results

Figure 2 reports the execution time of the kernel in Listing 6 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 consecutive 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 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.

5.1.3 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 consecutive 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.

5.2 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 (we found EHC implementation very slow and hence we avoid further discussion on this.):

i OpenACC without *cache* directive,

Table 2 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

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 4. Under RBC and RBI, we use *kernel arguments* as the default method for finding sharing width (discussed in Section 4.2.2) and we use *private* as the default method for renewing cache scope (discussed in Section 4.2.3).

Below we first compare development efforts of these four implementations. Next we compare performance of these implementations. Then we investigate how these implementations utilize GPU resources, e.g. register file and software-managed cache. Finally, we investigate the impact of alternative optimizations on the speedup.

5.2.1 Development Effort

We wrote all versions of GEMM and Jacobi. For N-Body Simulation, we used the CUDA version available in GPU Computing SDK [NVIDIA2017b] 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 2 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. As reported, OpenACC plus cache can be obtained by modifying 3 to 10 lines of the baseline OpenACC version.

5.2.2 Performance

GEMM: Cache-based OpenACC implementations 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 (without cache). A similar trend can be observed under different input sizes. RBI outperforms OpenACC by nearly 2.4X and performs very close to CUDA. 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* [NVIDIA2017a], 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

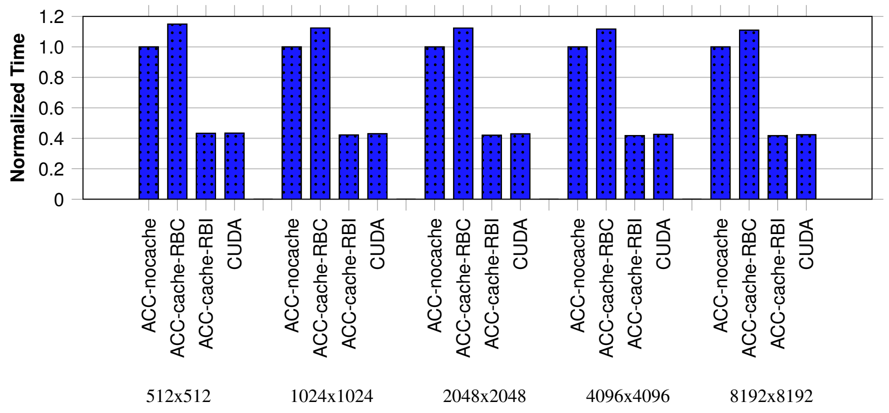

Figure 3: Comparing performance of four GEMM implementations under different matrix sizes. For each bar group, bars from left to right represent OpenACC without cache directive, OpenACC with cache directive implemented using RBC, OpenACC with cache directive implemented using RBI, and CUDA.

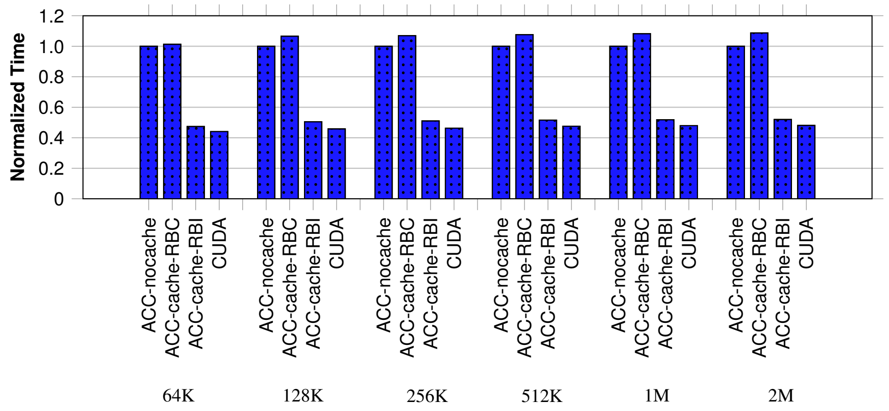

Figure 4: Comparing performance of four N-Body simulation implementations under different number of bodies.

too). 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 2.67X 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 6.

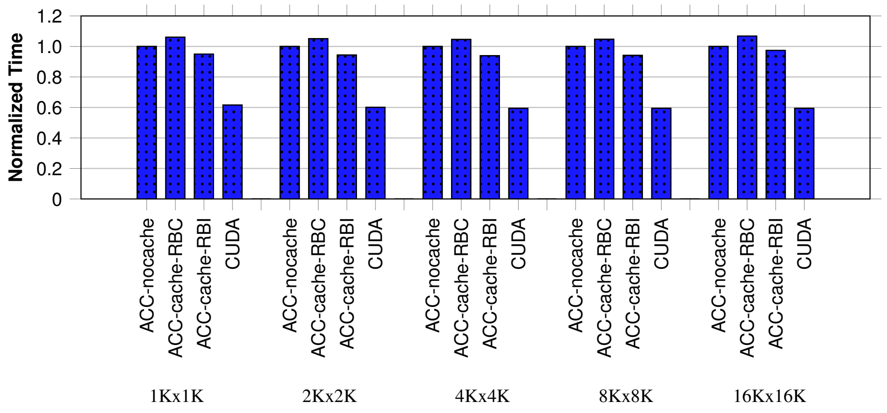

Figure 5: Comparing performance of four Jacobi iterative method implementations under different matrix sizes.

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 92%-111%. While RBI performs very close to CUDA, there is still a gap between them (8-10%). 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.

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 3-6%. 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.

Table 3 Comparing occupancy of OpenACC without cache, OpenACC plus cache (RBC and RBI), and CUDA.

	GEMM	N-Body	Jacobi
OpenACC-nocache	100% (24, 0)	100% (32, 0)	100% (16, 0)
OpenACC-cache-RBC	75% (33, 4KB)	75% (30, 8KB)	100% (21, 1.2KB)
OpenACC-cache-RBI	100% (30, 4KB)	75% (30, 8KB)	100% (18, 1.2KB)
CUDA	100% (30, 4KB)	100% (32, 4KB)	100% (11, 1.2KB)

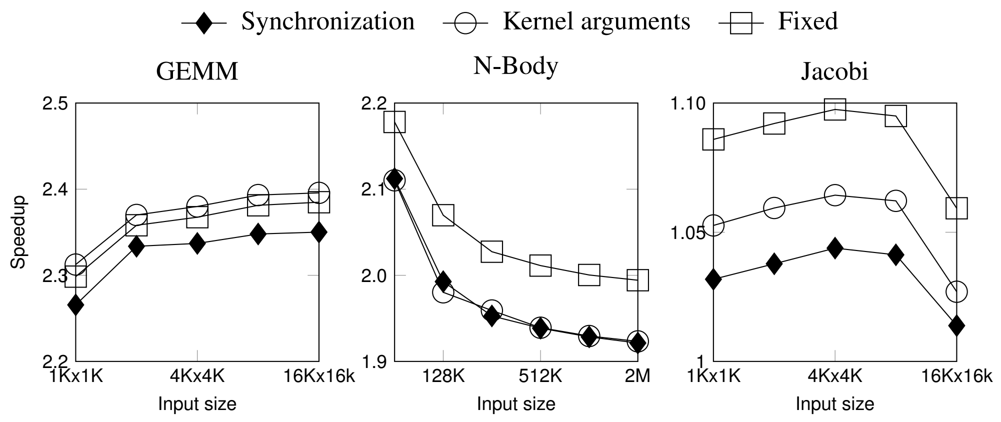

Figure 6: Comparing speedup from different finding sharing width methods. Numbers are normalized to the baseline OpenACC without using the cache directive.

5.2.3 Occupancy

Table 3 reports CUDA Occupancy of different implementations of the test cases discussed in Section 5.2.2. The table reports occupancy in percentage and, within the parentheses, the first number reports registers used per thread and the second number report the size of *shared memory* used per thread block. All implementations have the same thread block size: 256 under N-Body and 16 by 16 under GEMM and Jacobi. Occupancy is 100% in most cases, meaning that GPU is able to run up to 2048 threads per Streaming Multiprocessor. There are three cases where the occupancy is below 100%. RBC implementation of GEMM uses extra registers and that explains why occupancy drops below 100%. The size of cache after cache sharing is overestimated under RBC and RBI implementations of N-Body. This has lowered down the occupancy to 75%.

5.2.4 Implementation Alternatives

In Section 5.2, we reported performance of RBI and RBC under *kernel arguments* method of finding sharing width (discussed in Section 4.2.2) and *private* method of renewing cache scope (discussed in Section 4.2.3). In this section we investigate performance of RBI under alternative methods for finding sharing width and renewing cache scope (very similar discussion applies to RBC as well.)

Find Sharing Width: We compare speedup from three alternative methods for finding sharing width (*kernel arguments*, *synchronization*, and *fixed*), under three test cases introduced earlier (GEMM, N-Body, and Jacobi). *Fixed* method simply sets the sharing

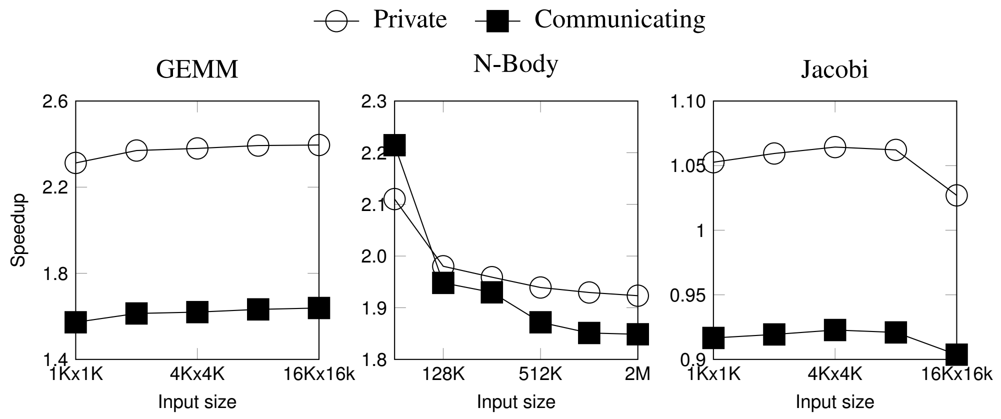

Figure 7: Comparing speedup from different renewing cache scope methods. Numbers are normalized to the baseline OpenACC without using the cache directive.

width to the thread block size. *Kernel arguments* method uses a control-flow statement per dimension and sets the sharing width either to the thread block size or a pre-calculated number (obtained from kernel arguments). *Synchronization* method performs one reduction per dimension of subarray to find sharing width. As reported in Figure 6, *fixed* method performs fastest. Although *fixed* method is the fastest, it is not generally applicable. This is because compiler may not be able to statically guarantee that the total number of loop iterations is multiple of thread block size. If this is the case, *kernel arguments* method can be used instead of *fixed* method. We found that the performance gap between *fixed* and *kernel arguments* is 3-5%. *Synchronization* method performs slowest under all test cases as reported in Figure 6 and performs up to 3% slower than *kernel arguments*. Reductions slow down performance of *synchronization* significantly for multi-dimensional subarrays. This is the case in GEMM and Jacobi that use two-dimensional subarrays. In N-Body, however, one-dimensional subarray is being used and *synchronization* method performs close to *kernel arguments*.

Renew Cache Scope: We compare speedup from two alternative methods of renewing cache scope (*communicating* and *private*), under three test cases introduced earlier (GEMM, N-Body, and Jacobi). *Communicating* method shares cache pointers among threads of the thread block and calculates the new scope collaboratively. Slow down under *communicating* method is incurred by thread block synchronizations and read/writes from *shared memory*. *Private* method, however, locally calculates the new cache scope according to the equations proposed in Section 4.2.3 and avoids debilitating inter-thread communications. As shown in Figure 7, *private* method outperforms *communicating* method under all test cases, except under smallest dataset of N-Body. In this case, the number of parallel threads is relatively low and GPU cores complete inter-thread communication very fast (since synchronization instructions are infrequently hindered by other instructions [Liu2016]). This makes *communicating* method faster than *private* method in this case. Overall *private* method outperforms *communicating* method by up to 47%.

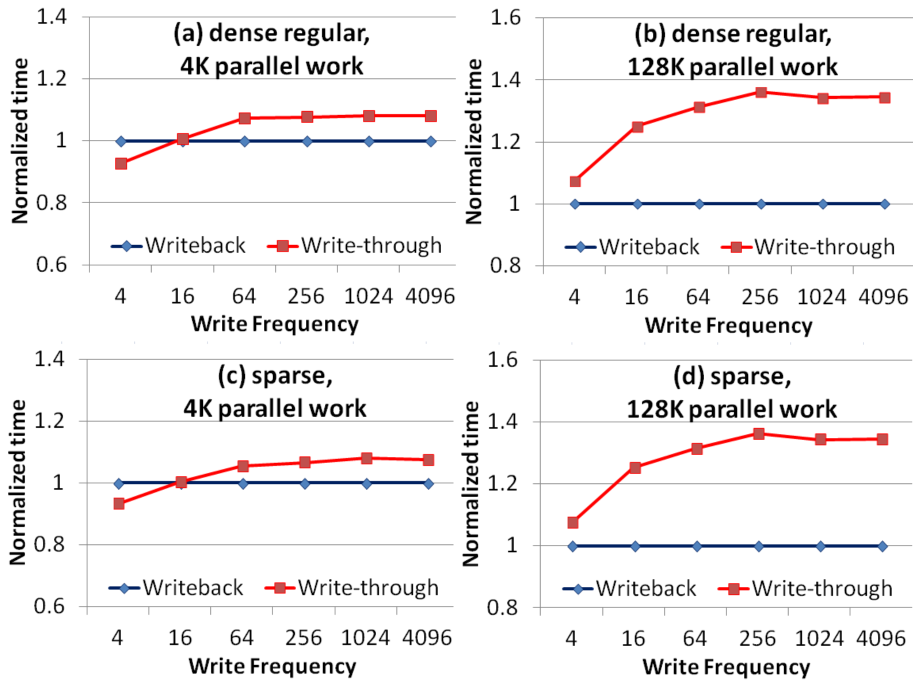

Figure 8: Comparing execution time of kernel under various shared memory configurations.

5.3 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 consecutive iterations write consecutive 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 8 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 is massive in size (e.g. 128K work), write-back is faster than write-through (Figure 8b and 8d). 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 is small in size and write frequency is low (e.g. left side of Figure 8a and 8c), write-through outperforms write-back. For example in Figure 8a, 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

Table 4 Performance improvement from RBI over the baseline OpenACC (without cache).

	Tesla K20c	Quadro K600
GEMM	238.0%	255.1%
N-Body	198.0%	211.4 %
Jacobi	6.4%	2.5%

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.

5.4 Performance Portability

Performance portability is one of the most important motivations of using OpenACC directives. In this paper, we focused on devising efficient implementation of the cache directive on the most commonly used platform [Norman2015; Bonati2015; Markidis2015], NVIDIA GPUs. Intuitively, we believe very similar optimization strategies can be followed on other similar architectures, e.g. AMD GPUs [AMD2012], to devise an efficient implementation of the cache directive. Discussing optimization strategies on different platforms is beyond the scope of this paper.

To show the performance portability across NVIDIA GPUs, here we evaluate our implementation on a different NVIDIA GPU, Quadro K600. In Table 4 we report performance improvement from RBI implementation over the baseline OpenACC (without cache) under three benchmarks: GEMM, N-Body, and Jacobi. We limit the evaluations to single dataset per benchmark (largest dataset that could fit in the memory of Quadro K600). For RBI configuration, we assume *private* method for renewing cache scope (Section 4.2.3) and *kernel arguments* method for finding sharing width (Section 4.2.2). As shown in the table, improvements are very close. Improvements are slightly larger under Quadro K600 for GEMM and N-Body benchmarks. This can be explained by the difference in the memory bandwidth of Quadro K600 and Tesla K20c. DRAM memory bandwidth of Quadro K600 is 29 GB/s which is 7.1 times lower than the bandwidth of Tesla K20c (208 GB/s). Accordingly, Quadro K600 is more sensitive to the techniques that optimize memory accesses. The cache directive is an example of these techniques and returns higher performance improvement when the memory bandwidth is throttled (e.g. Quadro K600).

6 Discussion

6.1 EHC in CUDA

In EHC, tag and data arrays should be kept consistent. This limits cache sharing and generally 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.

6.2 *Optimizing RBC*

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 [Fung2007]. 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 [NVIDIA2017c]). 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.

6.3 *Alternative cache targets*

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.

6.4 *Explicit mapping*

OpenACC API accepts hints from the programmer to explicitly specify the mapping of loop iterations to different thread blocks (*gang* clause) or the same thread block (*worker* and *vector* clauses). In this case, the compiler should generate a specific mapping of parallel loops to CUDA thread hierarchies, forced by *gang*, *vector*, and *worker* clauses. This can limit the range of compiler optimizations in sharing the cache space among threads. Generally, as long as the mapping enforced by the clauses is a *valid configuration* and *does not have conflict* with the outcome of the compiler pass we propose in Section 4.2.1, the compiler proceeds and exploits the sharing opportunity. Invalid configuration is created when the sharing range is larger than the CUDA shared memory size. This can be enforced by *vector* and *worker* clauses that map loop iterations to threads of one thread block and change the thread block size across *x* and *y* dimensions, respectively. The conflict mostly occurs when *gang* clause is used. *gang* clause asks the compiler to map each iteration to a thread block. This can have conflict with the compiler pass we presented in Section 4.2.1, 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 *gang* loop, or even ignoring the sharing optimization, in the worst case.

Alternative cache implementations 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. 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.

6.5 Cache Coherency

As stated by OpenACC specification [OpenACC2013; OpenACC2015], it is possible to write an accelerator parallel/kernels region that produces inconsistent numerical results. This is because some accelerators (e.g. GPUs) implement weak memory model. In weak memory model, memory coherency is not supported between operations executed by different threads nor between subsequent operations of a single thread, unless operations are separated by an explicit memory fence. This is the programmer's task to ensure the correctness of the application is not compromised by the weak memory model. Similarly, since variables and subarrays listed in the cache directive are part of the memory hierarchy, we assume weak memory model for the cache directive as well.

We explain the behavior of our implementation of the cache directive under this weak memory model under two major scenarios: i) one thread has the copy, the thread writes to the copy, and all threads attempt to read and ii) multiple threads have copies, multiple threads write to their copy, and all threads attempt to read. The behavior is summarized in Table 5. The behavior is very similar to CUDA behavior when multiple threads attempt to write to the same memory location asynchronously. Here we assume write-back policy for the cache implementation.

The first scenario assumes that one thread has a copy of a data in its cache, and no other thread has a copy of this data. Write operations to this local copy (by the thread) are immediately visible to the thread (marked as #1 in the table). As the thread leaves the cache region, the global memory will be updated by the latest copy available in the cache. While the kernel is running, threads that complete their read access before this update read the old value. Threads that complete their read access after this update read the new value. This behavior is marked as undefined in the table (#2). Once the kernel runs all the threads and completes its execution, the global memory has the new value rewritten by the thread (#3 in the table).

The second scenario assumes that multiple threads have copies of a data in their cache and one or more threads write their local copy. Since our implementation shares the cache among a range of threads (thread block), write operations from other threads to their local copy is visible to the range of threads that share the cache. Accordingly, a thread might read the value written by itself or other threads (#4 and #5 in the table). Our implementation guarantees that one write from the threads updates the cache, but the thread that updates the cache is unspecified. Similar to the first scenario, once a thread leaves the cache region, the global memory will be updated by the latest copy that is in the cache. Once the kernel

Table 5 Behavior of our weak memory model cache directive implementation under two scenarios: one write multiple reads and multiple writes multiple reads.

Scenario	During the kernel		After the kernel
	Seen by threads that have a copy	**Seen by other threads**	**Seen by all threads**
Only one thread has a copy and the thread writes to the copy	(#1) new value	(#2) undefined: old value or new value	(#3) the new value
Multiple threads have copies and one or more threads write to the copy	(#4) undefined: old value or new value from any copy	(#5) undefined: old value or new value from any copy	(#6) undefined: the new value from one unspecified thread

completes its execution, the global memory has the new value written by a thread, but again the thread is unspecified.

Overall, here the behavior for the cached and uncached data is the same. In either case, if multiple threads write to the same memory location, one thread successfully updates the global copy, but that thread is unspecified. Accordingly, future reads from that location might return undefined value.

7 Related Work

Reyes et al. [Reyes2012] 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. [Tian2013] presented an OpenACC implementation built in OpenUH [Liao2007]. Using OpenUH, they evaluated the performance of several alternatives in mapping loop iterations to GPU parallel threads. Lee and Vetter [Lee2014] 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. [Hoshino2014] 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. [Herdman2012] 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%.

8 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 i) sharing the cache among several parallel threads is essential to have a robust performance and ii) 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

[AMD2012] AMD Inc., "AMD Graphics Cores Next (GCN) Architecture," Available: https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf Last visited: 6 Apr. 2017.

[Bonati2015] C. Bonati, E. Calore, S. Coscetti, M. D'elia, M. Mesiti, F. Negro, S. F. Schifano, and R. Tripiccione, "Development of Scientific Software for HPC Architectures Using Open ACC: The Case of LQCD," in Proceedings of 2015 IEEE/ACM 1st International Workshop on Software Engineering for High Performance Computing in Science, Florence, 2015, pp. 9-15.

[Fung2007] 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

[Herdman2012] J. Herdman, W. Gaudin, S. McIntosh-Smith, M. Boulton, D. Beckingsale, A. Mallinson, and S. Jarvis, "Accelerating hydrocodes with openacc, opecl and cuda," in Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, Nov 2012, pp. 465-471.

[Hoshino2014] 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

[Lashgar2015] 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

[Lashgar2014] 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

[Lee2014] 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

[Liao2007] 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

[Liu2016] Y. Liu, Z. Yu, L. Eeckhout, V. J. Reddi, Y. Luo, X. Wang, Z. Wang, and C. Xu, "Barrier-Aware Warp Scheduling for Throughput Processors," In Proceedings of the 2016 International Conference on Supercomputing (ICS '16). ACM, New York, NY, USA, Article 42, 12 pages. [Online]. Available: https://doi.org/10.1145/2925426.2926267

[Markidis2015] S. Markidis, J. Gong, M. Schliephake, E. Laure, A. Hart, D. Henty, K. Heisey, and P. Fischer, "OpenACC acceleration of the Nek5000 spectral element code," The International Journal of High Performance Computing Applications, vol. 29, issue 3, pp 311-319, Mar. 2015.

[NVIDIA2017a] NVIDIA Corp., "Profiler's user guide: nvprof," Available: http://docs.nvidia.com/cuda/profiler-users-guide/#nvprof-overview Last visited: 19 Jan. 2017.

[NVIDIA2017b] NVIDIA Corp., "CUDA Downloads," Available: https://developer.nvidia.com/cuda-downloads Last visited: 19 Jan. 2017.

[NVIDIA2017c] NVIDIA Corp., "CUDA C Programming Guide," Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/ Last visited: 19 Jan. 2017.

[Norman2015] M. Norman, J. Larkin, A. Vose, and K. Evans, "A case study of CUDA FORTRAN and OpenACC for an atmospheric climate kernel," Journal of Computational Science, vol. 9, pp. 1-6, Jul. 2015.

[OpenACC2015] "The OpenACC Application Programming Interface Version 2.5," Available: http://www.openacc.org/content/openacc-2-5-final-specification.

[OpenACC2013] "The OpenACC Application Programming Interface Version 2.0," Available: http://www.openacc.org/sites/default/files/OpenACC%202%200.pdf

[Reyes2012] R. Reyes, I. Lopez-Rodriguez, 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

[Tian2013] 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, pp. 105-120.