Employing Software-Managed Caches in OpenACC: Opportunities and Benefits

Ahmad Lashgar, University of Victoria
Amirali Baniasadi, University of Vicoria

The OpenACC programming model has been developed to simplify accelerator programming and improve development productivity. In this paper, we investigate the main limitations faced by OpenACC in harnessing all capabilities of GPU-like accelerators. We build on our findings and discuss the opportunity to exploit a software-managed cache as: i) a fast communication medium and ii) a cache for data reuse. To this end, we propose a new directive and communication model for OpenACC. Investigating several benchmarks, we show that the proposed directive can improve performance up to 2.54X, and at the cost of minor programming effort.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**] Language Constructs and Features

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Accelerator, OpenACC, Software-managed cache, CUDA

ACM Reference Format:

Ahmad Lashgar and Amirali Baniasadi, 2015. Employing Software-Managed Caches in OpenACC: Opportunities and Benefits. ACM Trans. Model. Perform. 1, 1, Article 1 (December 2015), 34 pages.
DOI = 10.1145/2798724 http://doi.acm.org/10.1145/2798724

1. INTRODUCTION

In recent years, developers have introduced several programming models to simplify accelerator programming [Thies et al. 2002], [Mark et al. 2003], [Kapasi et al. 2003], [Buck 2004], [AMD Incorporation 2007], [Nickolls et al. 2008]. CUDA [Nickolls et al. 2008] which is a pioneer in this category, is a platform oriented model and can only be executed on a specific range of accelerators. OpenCL [The Khronos Group 2015] was introduced to address this limitation. Although these programming models are highly flexible and allow developers to achieve peak performance, developing an application using such models is not a trivial task.

The OpenACC programming model [OpenACC 2015] was developed to simplify accelerator programming. OpenACC exploits a directive-based API and allows developers to offload massively parallel computation regions of code to the accelerator. A previous work [Wienke et al. 2012] shows that using

Author's address: Ahmad Lashgar, Electrical and Computer Engineering Department, Room 220, Engineering Lab Wing Building, University of Victoria, Victoria, BC, Canada; email: lashgar@uvic.ca; Amirali Baniasadi, Electrical and Computer Engineering Department, Room 441, Engineering Office Wing Building, University of Victoria, Victoria, BC, Canada; email: amiralib@uvic.ca;

DOI 10.1145/2798724 http://doi.acm.org/10.1145/2798724

OpenACC can save up to 6.5X development effort (at the expense of a high performance gap.) We found this saving to be 5.82X in our benchmarks.

In this study, we identify and address main programming limitations in OpenACC. To this end, we compare several highly-optimized CUDA benchmarks (from Rodinia benchmark suite [Che et al. 2009] and CUDA SDK [NVIDIA Corp. 2015a]) to their OpenACC implementation. We show that the performance gap between OpenACC and CUDA implementation is significant when the CUDA version is optimized to take advantage of accelerators' software-managed cache (SMC) (shared memory of CUDA or local memory of OpenCL). We analyze CUDA implementations and identify three SMC advantages missing from OpenACC implementations: i) regular off-chip DRAM access resulting in fetching data in a well-coalesced pattern with no cost at SMC, ii) data reuse, which reduces off-chip traffic, and iii) inter-thread communication to facilitate fast on-chip communication among threads and to reduce global synchronizations [Garland et al. 2008].

OpenACC cannot take advantage of the above benefits as its abstractions do not allow programmers to harness SMC. As we show in this paper, this restriction can impose significant performance cost. Our goal in this work is to address this challenge and enhance performance in OpenACC implementations by making employing SMC possible. To this end, we propose a new directive, referred to as fcw (Fetch, Communication, and Writeback). We show that employing fcw can enhance performance by 2.54X, at the cost of minimal programming effort.

In summary, we make the following contributions:

—We measure and analyze the performance gap between OpenACC and CUDA implementations. We analyze OpenACC and CUDA implementations and identify the reasons behind the significant gap (up to 2.8X).
—We discuss the challenges in integrating SMC in OpenACC and propose *fcw*, to overcome the challenges. fcw reduces DRAM accesses frequency and introduces a new communication model, referred to as inter-iteration communication. This allows loop iterations to communicate through the fast on-chip cache, instead of global memory.
—We overview fcw in a simple reduction case study. We show how the programmer can use fcw and how compiler can translate the fcw code in the CUDA code, exploiting shared memory.
—We evaluate fcw under a wide set of benchmarks. First, we show how the highly-optimized CUDA version exploits SMC to minimize global communications. Then, we discuss how OpenACC limitations affect performance. Finally, we exploit fcw to improve performance in the OpenACC version.
—We discuss the limitations of cache directive and compare the performance of fcw and cache directives.

The remainder of this paper is organized as follows. In Section 2, we review CUDA and OpenACC. In Section 3, we identify the key issues affecting performance in OpenACC. In Section 4, we introduce fcw for OpenACC API. In Section 5, we explain a case study for exploiting fcw in OpenACC and translating fcw to CUDA. In Section 6, we present our methodology. In Section 7, we evaluate performance under fcw and provide better understanding. In Section 8, we discuss general applicability and performance issues related to fcw. In Section 9, we review related work. Finally, in Section 10, we present concluding remarks and future work.

2. BACKGROUND

2.1 CUDA Model

In CUDA [Nickolls et al. 2008], an application is composed of host and device codes. The host code executes on CPU and the device code executes on system's accelerator. The host controls the operations of the device through procedure calls to CUDA API. CUDA allows programmers to explicitly allocate memory on device and transfer data between the host and the device. The device obtains the device code from kernel and executes it by thousands of light-weight threads, in SIMT style [Lindholm et al. 2008]. All threads share common off-chip DRAM memory or global memory. In software, threads are grouped into coarser scheduling elements, referred to as the thread block. Threads within the same block execute concurrently and communicate through a fast, per-block, on-chip software-managed cache, referred to as shared memory. Shared memory is much faster than global memory; e.g. under GTX 280, the latency of global memory and shared memory are 440 and 38 core cycles, respectively [Wong et al. 2010]. We found these numbers to be 320 and 48 core cycles under NVIDIA Tesla K20c.

2.2 OpenACC Model

OpenACC API introduces a set of compiler directives, library routines, and environment variables to offload a region of code from the CPU to the system's accelerator [OpenACC 2015]. We refer to this region as the accelerator region. In a nutshell, OpenACC has two classes of directives: i) data management and ii) parallelism control. Each directive has clauses providing finer-grain control. Data management directives perform data allocation on the accelerator, data transfer between the host and the accelerator, and passing pointers to the accelerator. Parallelism control directives allow the programmer to mark regions of code, usually work-sharing loops, intended to run in parallel on the accelerator. They also control parallelism granularity, variable sharing/privatization, and variable reduction. OpenACC introduces four terms in loop parallelism: gang, worker, vector, and thread. In CUDA terminology, these terms best map to kernel, thread block, warp, and thread, respectively.

Listing 1 OpenACC matrix multiplication.

```
#pragma acc data copy(a[0:LEN*LEN],b[0:LEN*LEN],c[0:LEN*LEN])
#pragma acc kernels
#pragma acc loop independent
for(i=0; i<LEN; ++i){
#pragma acc loop independent
 for(j=0; j<LEN; ++j){
  float sum=0;
  for(l=0; l<LEN; ++l) sum += a[i*LEN+l]*b[l*LEN+j];
  c[i*LEN+j]=sum;
 }
}
```

Listing 2 CUDA matrix multiplication.

```
__global__ void matrixMul(int *a, int *b, int *c, int len){
 int i=threadIdx.x+blockIdx.x*blockDim.x;
 int j=threadIdx.y+blockIdx.y*blockDim.y;
 for(int l=0; l<len; ++l) sum=a[i*len+l]*b[l*len+j];
 c[i*len+j]=sum;
}
int main(){
  ...
 bytes=LEN*LEN*sizeof(int);
 cudaMalloc(&a_d, bytes);
 cudaMalloc(&b_d, bytes);
 cudaMalloc(&c_d, bytes);
 cudaMemcpy(a_d, a, bytes, cudaMemcpyHostToDevice);
 cudaMemcpy(b_d, b, bytes, cudaMemcpyHostToDevice);
 dim3 gridSize(LEN/16,LEN/16), blockSize(16,16);
 matrixMul<<<gridSize,blockSize>>>(a_d,b_d,c_d,LEN);
 cudaMemcpy(c, c_d, bytes, cudaMemcpyDeviceToHost);
  ...
}
```

2.3 Matrix-Matrix Multiplication Example

Listing 1 and 2 illustrate a simple matrix-matrix multiplication in OpenACC and CUDA, respectively. Ignoring the directive lines, Listing 1 shows the baseline serial multiplication of *a[]* and *b[]*, storing the result in *c[]*. Each matrix is *LEN*LEN* in size. The outer loops iterated by *i* and *j* variables can be executed in parallel. Listing 1 shows how these loops can be parallelized using OpenACC. In this code, *kernels* directive marks a region intended to be executed on the accelerator. *loop* directive guides the compiler to consider the loop as a parallel work-sharing loop. Programmers can control the parallelism using kernels and loop directives. As an example of parallelism control, the independent clause is used to force the compiler to parallelize the loop. This clause overwrites the compiler's auto-vectorization and loop dependency checking. In Listing 1, data clauses hint the compiler to copy *a[]*, *b[]*, and *c[]* arrays from the host to the accelerator, and copy them out from the accelerator to the host. For each array, the [*start*:*n*] pair indicates that *n* elements should be copied from the *start* element of array. Listing 2 shows how the parallelization can be exploited in CUDA. *__global__* indicates the declaration of kernel code. Parallel threads execute the kernel and operate on different matrix elements, based on their unique indexes (*i* and *j*). Inside the host code, device memory is allocated for *a[]*, *b[]*, and *c[]*, keeping the pointer in *a_d[]*, *b_d[]*, and *c_d[]*, respectively. Then, input matrices are copied into device memory. Then, a total of *LEN*LEN* light-weight accelerator threads are launched on the device to execute *matrixMul* kernel. After kernel completion, the resulting matrix *c_d[]* is copied back to the host memory. As presented in Listing 1, OpenACC significantly reduces the accelerator programming effort in comparison to CUDA. OpenACC hides low-level accelerator-related code from the programmer and provides a unified view over both host and accelerator code.

2.4 Terminology

We define and use the following terms in the remainder of paper. *Parallel work-sharing loop* or simply *parallel loop* refers to a loop which is marked by OpenACC to be executed on the accelerator. *Parallel iterations* refer to the iterations of a parallel loop. We assume that each parallel iteration is mapped to one light-weight accelerator thread. Therefore, we use the terms parallel iteration and thread interchangeably. We refer to consecutive iterations of a parallel loop as *neighbor iterations* or *consequent iterations*. In the sequence of parallel iterations, every parallel iteration has neighbor iterations. The definition of neighbor iterations ignores the size of neighborhood and can be of any range. For example, if a parallel loop iterates from 1 to N by one step, parallel iterations indexed by 8, 9, 10, 11, and 12 are neighbor iterations.

3. MOTIVATION

In this section, we compare a set of OpenACC applications to their optimized CUDA version. Our goal is to identify OpenACC's programming limitations resulting in the performance gap between OpenACC and CUDA performance. See Methodology section for applications, compilers, and hardware setup.

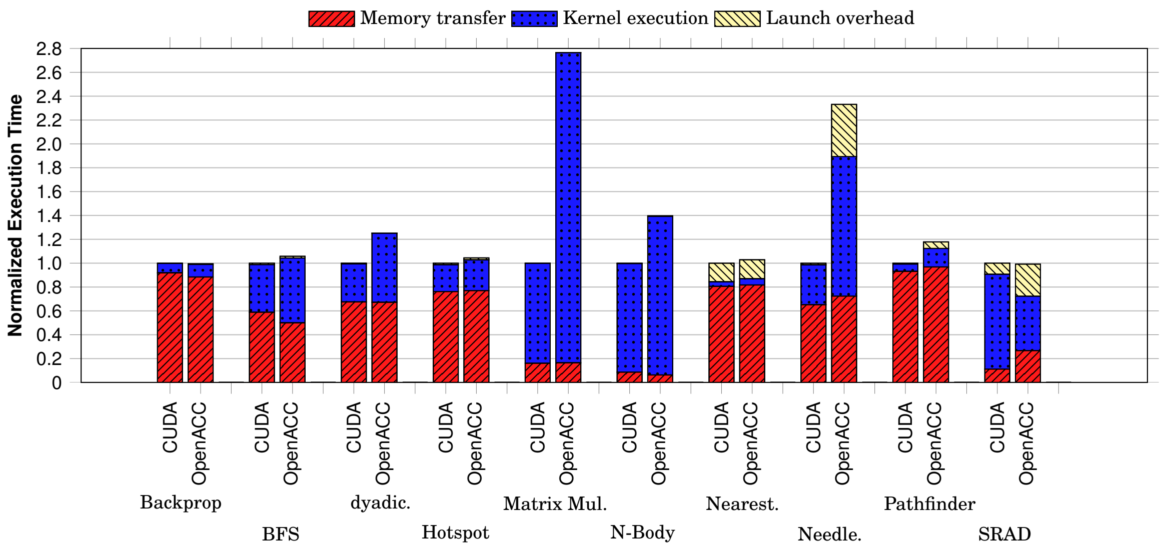


Fig. 1. Comparing the execution time of OpenACC to highly-optimized CUDA implementations. Each bar shows the duration of time that the application spends on memory transfer, kernel execution, and kernel launch overhead.

3.1 Performance Comparison

Figure 1 reports the execution time for OpenACC applications, compared to their CUDA version. The figure reports the breakdown of time spent on the accelerator; kernel launch (launch), kernel execution (kernel), or memory transfer between host and accelerator (memory). Kernel launch time includes the time spent on setting kernel arguments and launching the kernel on the accelerator. In most cases, CUDA's kernel launch/execution portion is shorter than OpenACC. Also, memory transfer times are comparable on both CUDA and OpenACC. There are exceptions where OpenACC memory transfers are faster (e.g. BFS) or kernel time of CUDA and OpenACC are equal (e.g. Nearest.). Below we investigate the differences between CUDA and OpenACC.

3.2 Investigating Performance Gap

In this section, we discuss applications separately providing insight into why CUDA and OpenACC implementations presented in Figure 1 have different kernel launch, kernel execution, and memory transfer times.

Back Propagation. Back Propagation (Backprop) is a machine-learning algorithm used to train the weights in a three-layer neural network. In both OpenACC and CUDA versions, there are six back-to-back serial operations where the output of each stage is fed to the immediate next stage as input. Each stage can be performed in parallel on the accelerator. OpenACC and CUDA versions offload the first and last stages to GPU.

OpenACC implementation performs faster memory transfers and slower kernel launches/executions, compared to CUDA. This is explained by the difference between CUDA and OpenACC in implementing the first stage, which is similar to reduction. Both CUDA and OpenACC reduce the array in two levels; first on the GPU and the second on the CPU. However, OpenACC uses larger thread blocks which lower the memory transfer time, but increase the kernel execution time. This explains why CUDA has faster kernel execution and slower memory transfer time.

BFS. BFS visits all the nodes in the graph and computes the visiting cost of each node. Each node is visited only once. Parallel threads of a kernel visit the nodes belonging to the same graph depth concurrently and the algorithm traverses through the depth iteratively. The operation stops once there is no child to visit.

Compared to the CUDA version, the OpenACC version of BFS spends less time on memory transfers. This can be explained by the fact that the OpenACC version performs data initializations on the GPU. However, the CUDA version initializes the inputs on the host and transfers the inputs to GPU. Compared to the CUDA version, OpenACC spends more time on kernel execution, since it forces a debilitating reduction on a global variable. The global variable is a boolean indicating whether there remains more nodes to visit or not. CUDA avoids global reduction by initializing the variable to FALSE on the host and imposing a control-flow divergent in the kernel to guard the global variable from FALSE writes (allowing TRUE writes).

Dyadic Convolution. Dyadic Convolution (dyadic.) is an algebra operation calculating the XOR-convolution of two sequences. The OpenACC implementation parallelizes output calculations, where each thread calculates one output element. Although this implementation is fast to develop, it exhibits a high number of irregular memory accesses. To mitigate irregular memory accesses, the CUDA version uses Fast Walsch-Hadamard Transformation (FWHT) for implementing dyadic convolution (as described in [Arndt 2011]).

As reported in Figure 1, both OpenACC and CUDA versions spend almost the same amount of time on memory transfers. While the CUDA version launches several kernels, OpenACC launches only one kernel. Kernel launch time share, however, is negligible in the time breakdown. In CUDA the kernels' execution time is 82% faster than OpenACC. This is due to the fact that the CUDA version uses FWHT to mitigate irregular memory accesses. Although OpenACC can implement dyadic convolution using FWHT, the same FWHT algorithm used in CUDA cannot be implemented in OpenACC. CUDA FWHT uses shared memory to share intermediate writes locally between neighbor threads, which is not possible under OpenACC standard.

Hotspot. Hotspot simulates chip characteristics to model the temperature of individual units. At every iteration, the algorithm reads the temperature and power consumption of each unit and calculates new temperatures. Although both OpenACC and CUDA spend the same amount of time on memory transfers, CUDA kernel is faster.

In Hotspot, the temperature of each unit depends on its power consumption and neighbors' temperatures. CUDA kernel exploits this behavior to localize the communication and reduce global memory accesses as follows. In CUDA, threads of the same thread block calculate the temperature of neighbor units. The CUDA version locally updates the new temperature of neighbor units using the threads of the same thread block. This local communication reduces the number of kernel launches used to synchronize the temperature across all thread blocks, explaining why the CUDA version performs faster kernel launches and comes with shorter execution time. In OpenACC, unlike CUDA, the software-managed cache cannot be exploited for local communication. Hence, in OpenACC there are higher number of global synchronizations and kernel launches, which in turn harms performance.

Matrix Multiplication. Matrix Multiplication (Matrix Mul.) performs multiplication of two 1024 by 1024 matrices. Both CUDA and OpenACC implementations use output parallelization, calculating each element of the output matrix in parallel. CUDA version is different from OpenACC as it processes input matrices tile-by-tile. By processing in tiles, CUDA version fetches the input tiles in few well-coalesced accesses into software-managed cache and shares the tiles among the threads of the same thread block.

While kernel launch and memory transfer times are nearly the same across CUDA and OpenACC, CUDA kernel time is much lower than OpenACC. CUDA version takes advantage of software-managed cache in two ways. First, CUDA version merges the required data of the thread block and fetches them once, minimizing redundant memory accesses across thread of the same thread block. Second, software-managed cache removes cache conflict misses, since the replacement policy is controlled by the programmer. Under OpenACC, although the threads have very high spatial locality, parsing the matrix row-by-row at a time highly pollutes the cache, returning high number of conflict misses. Also having multiple thread blocks per SM exacerbates this effect.

N-Body simulation. N-Body models a system of particles under the influence of gravity force. In each timestep, operations of O(N^2) complexity are performed (for a system of N particles) to calculate forces between all pairs of particles. Inherently, there are many redundant memory reads, since the mass and position information of each particle is fetched by other particles N-1 times to calculate its interaction with other particles.

OpenACC memory transfers are slightly faster than CUDA as OpenACC uses *float3* data type where applicable. CUDA, on the other hand, uses *float4* for all structures. Comparing kernel execution times, the CUDA version is much faster as it tiles the computations to reduce redundant memory reads [Nyland and Harris 2007]. CUDA exploits shared memory to share the particles among all threads of a thread block. In OpenACC, however, the redundant memory accesses are not filtered out by the software-managed cache. As reported, redundant memory accesses can degrade performance significantly.

Nearest Neighbor. Nearest Neighbor (Nearest.) finds the five closest points to a target position. The Euclidean distance between the target position and each of the points is calculated and the top five points with the lowest distance are returned. OpenACC and CUDA versions both calculate Euclidean distances for each point in parallel. OpenACC and CUDA versions spend about the same time on kernel launch, kernel execution, and memory transfer. This is explained by the similarity of parallelization methods applied in both OpenACC and CUDA.

Needleman-Wunsch. Needleman-Wunsch (Needle.) is a sequence alignment algorithm used in bioinformatics. In either CUDA or OpenACC, traverses a 2D matrix and updates the costs. Upon updating a new cost, four memory locations are read and one location is written.

Although both CUDA and OpenACC versions spend the same amount of time on memory transfers, CUDA kernel launch/executions are much faster than OpenACC kernels. The CUDA version fetches a data chunk of *costs* matrix into shared memory and traverses the matrix at the shared memory bandwidth. This mechanism comes with three advantages: i) filtering redundant global memory accesses by shared memory, ii) minimizing global communication by sharing intermediate results stored in the shared memory, iii) reducing the number of kernel launches and global communications. The fewer number of kernel launches explains why the launch time of CUDA is much less than OpenACC.

Pathfinder. In Pathfinder kernel, every working element iteratively finds the minimum of three consequent elements in an array. The CUDA version of Pathfinder performs two optimizations: i) finding the minimum by accessing the data from shared memory, and ii) sharing the updated minimum locally among neighbor threads for certain iterations and then reflecting the changes globally to other threads. Such local communications reduce the number of global synchronizations and kernel launches.

However, OpenACC's API is not flexible enough to allow the programmer exploit the shared memory in a similar way. Therefore neighbor threads in the OpenACC version do not communicate via shared memory. Therefore, each thread fetches the same data multiple times and threads communicate only through global memory. Communication through global memory is implemented through consequent kernel launches. This explains why OpenACC imposes higher kernel launch overhead.

Speckle reducing anisotropic diffusion. Speckle reducing anisotropic diffusion (SRAD) is an image processing benchmark performing noise reduction through partial differential equations iteratively. Compared to CUDA, the kernel time of OpenACC version is less. Three code blocks construct the computation iterative body of this benchmark: one reduction region and two data parallel computations. Our evaluation shows OpenACC version performs 5% slower than CUDA, upon executing two data parallel computations. However, OpenACC kernel time is lower than CUDA in executing the reduction portion. This is explained by the difference in reduction implementations. Our OpenACC framework performs the reduction in two levels: reducing along threads of thread block on GPU and reducing along thread block on CPU. In the CUDA version, however, reduction is performed by multiple serial kernel launches, all on the GPU. The OpenACC version spends less time on executing the kernel as part of the computation is carried on host. Meanwhile, performing two-level reduction imposes the overhead of copying intermediate data from GPU to CPU. This explains why the OpenACC version spends more time on memory transfers. The OpenACC version also spends more time on kernel launch as explained by the overhead of OpenACC API calls for finding the host pointers on the device. This benchmark has many data clauses which signifies this overhead[1].

3.3 Putting It All Together

As reported in the previous section, we observe a wide performance gap between CUDA and OpenACC when the CUDA version is optimized to exploit SMC. The CUDA version of N-Body and Matrix Multiplication benchmarks use SMC to reuse the data at the bandwidth of fast SMC. CUDA versions of Hotspot, Needleman-Wunsch, Pathfinder, and Dyadic Convolution use SMC to perform fast local communications. Exploiting SMC in these benchmarks not only reduces the kernel execution time, but also reduces the number of kernel launches, saving high kernel launch time.

According to our observations, lack of a flexible API for exploiting software-managed caches is the key limitation in OpenACC programming, widening the performance gap between OpenACC and CUDA. In this study, we focus on improving OpenACC's flexibility in exploiting software-managed cache. In the next section, we discuss the challenges in exploiting SMC in OpenACC.

3.4 Challenges in Integrating SMC in OpenACC

There are two main challenges in integrating SMC in OpenACC.

Communication. To perform communication through SMC, programmers should clearly understand the communicating elements and the available synchronization. In CUDA, communicating elements are threads of the same thread block. In OpenACC, however, there is no semantic for communication among iterations of parallel work-sharing loops. Communicating elements and available synchronizations should be introduced.

Data range identifying. To take advantage of SMC for data reuse, it is critical to identify a data range exhibiting temporal locality. This range can be best identified by the programmer. Through an interfacing notation, the programmer can pass this information to the compiler. The notation should be simple enough for the programmer, while still informative for the compiler for exploiting SMC.

In the next section, we propose fcw as a directive to address these challenges.

4. PROPOSED DIRECTIVE

In this section, we introduce a new OpenACC directive, referred to as *fcw* directive (fetch, communicate, and writeback). fcw is applied over a region of code and is only available within kernels/parallel regions. This directive serves as a compiler hint to fetch a data chunk into local software-managed cache (SMC), replace global memory accesses with SMC accesses, allow concurrent accelerator threads to communicate through SMC, and write the local data back to global memory.

[1]Currently, we use linked list for mapping host memory pointers to accelerator memory pointers. This overhead can be mitigated using hash tables.

The motivation behind fcw stems from the fact that consequent iterations of a parallel loop often share data. fcw allows neighbor iterations to share intermediate data. It allows fast communication between neighbor iterations through SMC, replacing costly global memory communication/synchronization. In addition, the compiler can mitigate redundant memory fetches by exploiting programmer's hints. The programmer specifies the data that is shared among consequent iterations. This information guides the compiler to fetch common data by few well-coalesced requests, preventing repeated global memory fetches, and reusing data at the SMC bandwidth.

fcw gives the programmer the option to specify the shared data by denoting the range of indexes that need to be cached in SMC. This range can be specified with respect to constant terms or even the induction variable of the outer parallel work-sharing loop. In either case, the data of neighbor iterations may construct a continuous memory chunk. The compiler generates a code to fetch this chunk from global memory to SMC. Future memory accesses within this chunk use SMC bandwidth, instead of global memory. Otherwise, if the range expression is derived from constant terms, the same data can be fetched/shared among all iterations of the loop.

A common scenario occurs when fcw is applied within a parallel loop where the range identifiers depend on the loop induction variable. In this case, the range of indexes for an iteration may overlap with the range of indexes evaluated on its neighbor iterations. The union of this overlapping data creates a consequent chunk of data. Building on this fact, fcw can be implemented efficiently, if the OpenACC implementation executes OpenACC applications over CUDA API. Under OpenACC to CUDA translations, neighbor iterations of a parallel work-sharing loop usually map to consequent threads of the same thread block [Tian et al. 2013]. In CUDA terms, a thread block is the collection of certain number of consequent iterations. Therefore, CUDA shared memory can serve efficiently as SMC to share data among the iterations of a parallel loop (or threads of the same thread block.)

In the remainder of this section, we define fcw's programming interface and present examples to clarify the notation.

4.1 Programming Interface

We introduce fcw by the following notation:

#pragma acc fcw SMC_TYPE(arrayName[PIVOT : BEFORE : AFTER])

where the parameters are described below.

arrayName is a variable name pointing to the memory location intended to be cached in SMC.

PIVOT, *BEFORE*, and *AFTER* are guides to identify the range of data required to be cached in SMC. PIVOT denotes the central index. Regardless of BEFORE and AFTER, the compiler generates a code to cache the element in the PIVOT position. BEFORE/AFTER denotes the number of elements before/after PIVOT that should be fetched into SMC. These parameters are best explained through an example, as we show in Table I. In multidimensional arrays, the tuple of {PIVOT, BEFORE, AFTER} can be replicated to point different dimensions of an array.

fcw can have different types which is specified by *SMC_TYPE* clause. Although all fcw types access the specified memory region at the bandwidth of SMC, they differ based on three further operations: i) initializing the SMC data from global memory (fetch), ii) making intermediate writes visible to neighbor iterations (communication channel), and iii) reflecting local changes back to global memory (writeback). Whether necessary to activate each of these three operations or not, there are eight possible fcw types available to programmer. The eight types combine all possible combinations of fetching data from global memory to SMC (or not), creating communication channels (or not), and writing back data to global memory (or not). We investigated several CUDA benchmarks to find the most common combinations. Based on the studied workloads, we found a subset of five combinations to be the most common among the eight: FETCH_ONLY, CHANNEL_ONLY, FETCH_CHANNEL, CHAN-

NEL_WB, and FETCH_CHANNEL_WB. In FETCH_ONLY, the compiler only fetches the data from global memory to SMC. In this case, no local communication channel is created and writeback are not performed. In CHANNEL_ONLY, the compiler neither fetches data from global memory to SMC nor writes it back to global memory after exiting the region. This mode is only useful for fast interchanging/sharing intermediate data among loop iterations through SMC. FETCH_CHANNEL is similar to CHANNEL_ONLY but also fetches/initializes the corresponding data from global memory to SMC. CHANNEL_WB is similar to CHANNEL_ONLY but also writes the final data back to global memory. FETCH_CHANNEL_WB performs all of the three operations. It fetches/initializes the data into SMC, creates a communication channel among neighbor iterations, allows intermediate writes to be visible to neighbor iterations, and writes the final value of the SMC back to global memory. CHANNEL_ONLY, FETCH_CHANNEL, CHANNEL_WB, and FETCH_CHANNEL_WB create a local communication channel and allow intermediate changes to SMC to be visible to all threads within the same thread block.

4.2 Communication Model

fcw allows fast communication through SMC. We refer to this new communication model as inter-iteration communication. Inter-iteration communication allows communication among iterations of a parallel work-sharing loop.

In this model, at the high-level programmer view, all parallel iterations are grouped into communicating thread blocks, in which every iteration belongs to only one communicating thread block. Iterations within the same communicating thread block can share data via SMC. To avoid data hazards, implicit synchronizations is performed after every write to the location marked as SMC. Also the number of iterations belonging to the same communicating thread block, or the communicating thread block size, can be configured.

CHANNEL_ONLY, FETCH_CHANNEL, CHANNEL_WB, and FETCH_CHANNEL_WB allow various communication patterns. CHANNEL_ONLY creates the basic communication medium. It constructs a communication medium within a communicating thread block using SMC bandwidth. Iterations can share intermediate results through this fast cache. FETCH_CHANNEL, unlike CHANNEL_ONLY, also fetches/initializes SMC from global memory. CHANNEL_WB is the same as CHANNEL_ONLY except it also writes back the final state into global memory. Finally, FETCH_CHANNEL_-WB is different from CHANNEL_WB, as it allows initializing SMC by fetching from global memory. Notice that FETCH_ONLY only allows data reuse at SMC bandwidth and does not construct a communication channel within a communicating thread block.

Since data hazards and race conditions may occur, sharing SMC among iterations of the same communicating thread block needs inter-iteration synchronization. Under inter-iteration communication, there are two kinds of synchronizations: implicit and explicit. For fcw types including FETCH operation, there is an implicit synchronization between iterations fetching data for a thread block. This is implicit and hidden from the programmer as the compiler generates the associated code automatically. Also fcw types that include CHANNEL operation contain implicit synchronization, which is forced by the writes to SMC. For the data chunk that is marked in fcw, the programmer can assume one barrier call before every individual memory write and one after it. We refer to this as implicit synchronization since it is performed implicitly by the compiler, hidden from the programmer. While implicit synchronization satisfies the need for synchronization, programmers can use explicit synchronization to control synchronization selectively. We add the following directive to OpenACC API, allowing manual inter-iteration synchronization: *#pragma acc fcw_barrier*. *fcw_barrier* synchronizes the iterations of the communicating thread block and assures that all previous SMC writes are completed. Notice that *fcw_barrier* can be implemented through CUDA thread block synchronization barriers.

Table I. Examples of determining range identifiers to direct the compiler for SMC.

Example	a pivot	a before	a after

`(a)`
`#pragma acc kernels loop`
`for(i=1; i<(size-1); ++i)`
` #pragma acc fcw FETCH_ONLY(a[i:0:0])`
` {b[i]=a[i];}`
	i	i	i

`(b)`
`#pragma acc kernels loop`
`for(i=1; i<(size-2); ++i)`
` #pragma acc fcw FETCH_ONLY(a[i:1:2])`
` {b[i]=a[i-1]+a[i]+a[i+2];}`
	i	i-1	i+2

4.3 Example

Table I illustrates two examples of configuring PIVOT, BEFORE, and AFTER parameters. In these examples, a is a pointer to a read-only array, targeted to be cached in SMC. As specified by the first parameter, fcw type is FETCH_ONLY. I(a) illustrates an example where the required data simply includes one element, pointed to by the loop induction variable. In this case, the pivot element is i (the loop induction variable) and fcw guides the compiler to fetch only pivot (i:0:0). In I(b), the pivot element is set to i and fcw directs the compiler to fetch a range from one element before the pivot to two elements after the pivot (i:1:2).

In these examples, the required data of consequent iterations construct a continuous memory chunk. For example in I(b), the required data of every consequent 256 iterations is 259 elements. Since the threads of the same thread block execute consequent iterations, using the information provided through fcw, the compiler can identify the data of each thread block and map the corresponding memory chunk into SMC. With a mapping that is invisible to the programmer, the compiler replaces every individual array access with an SMC access. The compiler also generates proper fetch and writeback, based on fcw type provided through SMC_TYPE.

5. CASE STUDY: REDUCTION

In this section, we review our automatic code generation through a sample. Our goal is to investigate the applicability of fcw in implementing basic parallel algorithms. Specifically, we present a reduction case study to show how fcw notation can be translated to CUDA to exploit shared memory. Listing 3 shows the fcw-enhanced OpenACC version. Listing 4 and 5 show the CUDA version translated from Listing 3. Listing 4 shows the host code and Listing 5 shows the accelerator code.

The code shown in Listing 3 calculates the reduction in several kernel launches. The outer data directive copies input arrays (*a[]* and *b[]*) to accelerator's memory before the data region and copies them back to host's memory after the region. Within the data region, *kernels* region is translated into a procedure call on accelerator. The kernel reduces every 256 elements (TILESIZE) of *a[]* and stores the summation into *b[]*. The outer loop iterates the kernel launches. In every iteration, the code swaps *a[]* and *b[]* pointers, shrinks the size of arrays by a factor of TILESIZE, and launches the kernel to reduce the new *a[]*. This process relapses until the size of *a[]* shrinks below TILESIZE, returning the final summation in *b[0]*.

In every kernel call, a total of P threads (equal to the current *a[]* size) are launched on the device. The fcw directive guides the compiler to fetch a tile of elements for each thread block into shared memory. Threads of the same thread block calculate the sum of these elements at the bandwidth of

SMC and write the final reduced value to *b[]*. The tile size is set to TILESIZE by *vector* clause of the loop directive.

Listing 3 Reduction in fcw.

```
uint TILESIZE=256, TILESIZEB=8;
#pragma acc data copy(a[0:N],b[0:N])
for(P=N; P>1; P=P/TILESIZE+(((P%TILESIZE)!=0)?1:0)){
  int *c=a; a=b; b=c;
  #pragma acc kernels present(a,b)
  #pragma acc loop independent vector(TILESIZE)
  for(i=0; i<P; i++){
    int tileid   =i>>TILESIZEB;
    int offsetid=i&(TILESIZE−1);
    int k=N;
    #pragma acc fcw FETCH_CHANNEL(a[i:0:0])
    {
      for(y=TILESIZE/2; y>0; y>>=1)
        if(( offsetid<y) && ((i+y)<P))
          a[i]=a[i]+a[i+y];
      if(i==(tileid*TILESIZE))
        b[ tileid ]=a[i ];
    }
  }
}
```

Listing 4 Host code of translation of fcw Reduction to CUDA.

```
#define TILESIZE 256
acc_create((void*)a,(N+0)*sizeof(int));
acc_create((void*)b,(N+0)*sizeof(int));
acc_copyin((void*)a,(N+0)*sizeof(int));
acc_copyin((void*)b,(N+0)*sizeof(int));
P = N;
for(P=N; P>1; P=P/TILESIZE+(((P%TILESIZE)!=0)?1:0)){
  int *c = a;  a = b;  b = c;

  /* kernel call  statement */
  {
    __generated_kernel_region_0
  <<<((abs((int)((P))−0))/(1))/TILESIZE+1,TILESIZE>>>
  ((int*)acc_deviceptr((void*)a),
         (int*)acc_deviceptr((void*)b),
         N, P);
  }
  cudaError err=cudaDeviceSynchronize();
}
acc_copyout((void*)a,(N+0)*sizeof(int));
acc_copyout((void*)b,(N+0)*sizeof(int));
```

Listing 4 shows the CUDA code obtained by automatic translation of Listing 3. data clauses are translated to OpenACC API calls, e.g. *acc_create*, *acc_copyin*, and *acc_copyout*. The code included in the kernel region of Listing 3 is translated to a CUDA kernel named *__generated_kernel_region_0*. For launching this kernel, the thread block size is derived from *vector* clause of the parallel loop. The number of thread blocks is the number of loop iterations divided by the thread block size.

Listing 5 shows the kernel code translated from the *kernels* region of Listing 3. At the beginning lines of the code (line #2), a variable is declared to obtain the global identifier of the thread along the parallel loop. This is followed by the declaration of loops' induction variable (line #3), shared memory space (line#6), and two pointers to locations in global memory (line #7 and #8). Loops' induction variables are declared within the kernel locally (allowing the compiler to reserve a register for each of them). The size of shared memory is the sum of thread block size (TILESIZE), number of elements before the PIVOT (BEFORE field in fcw directive of Listing 3, which is 0), and number of elements after the PIVOT (AFTER field in fcw directive of Listing 3, which is also 0). These are specified by the programmer via *vector* and *fcw clauses*, respectively. Two pointers, which are assigned later in the code (line #14 to #25), point to the start and end addresses of the global memory data chunk (*a[]*) which is mapped to the shared memory space (*__kernel_smc_var_data_a*) of each thread block. The start and end pointers point to the first and last threads of the thread block, respectively. The body of parallel loop is injected within the kernel code (line #9 to #46). At beginning code block, each thread calculates the value of the loop induction variable which corresponds to its thread's global identifier (line #9). Next, the *if* statement assures the parallel boundary and guards the loop body from extra threads (line #10). As shown in Listing 3, the fcw type is FETCH_CHANNEL. This guides the compiler to initialize the shared memory by fetching the data from global memory. FETCH_CHANNEL also guides to create a channel to allow communication among the threads of the thread block and make intermediate writes locally visible. The body of the loop consists of three code portions: fcw fetch (line #13 to #26), index precalculations (line #28 to #32), and fcw regions. fcw fetch calculates the boundary of the tile assigned to the thread block (line #14 to #18). Line #19 to #24 use parallel threads of the thread block to fetch a data chunk from global memory to shared memory. Compiler statically finds the unique memory indexes accessed within fcw region and injects a code to precalculate their mapping from global space to shared space at runtime (line #30 and #32). To create the communication channel within the fcw region, accesses to *a[]* are replaced by accesses to *__kernel_smc_var_data_a*. An intra-thread block synchronization call is injected before and after every write to SMC location, preventing race condition.

6. METHODOLOGY

Benchmarks. We use benchmarks from NVIDIDA CUDA SDK [NVIDIA Corp. 2015a] and Rodinia benchmark suite [Che et al. 2009]. NVIDIA CUDA SDK includes a large set of CUDA test-cases, each implementing a massively-parallel body of an application in CUDA efficiently. Each test-case also includes a serial C/C++ implementation. We developed an OpenACC version of these benchmarks over the serial C/C++ code. Rodinia is a GPGPU benchmark suite composed of a wide set of workloads implemented in C/C++. Originally, each of these benchmarks was implemented in CUDA and OpenCL parallel models. OpenACC implementation of the benchmarks has been added by a third-party [PathScale 2013]. We include Dyadic Convolution, Matrix Multiplication, and N-Body simulation from CUDA SDK and the remaining benchmarks from Rodinia.

OpenACC Compiler. We use our in-house framework, IPMACC [Lashgar et al. 2014], for compiling OpenACC applications. IPMACC translates OpenACC to either CUDA or OpenCL and executes OpenACC application over CUDA or OpenCL runtime (e.g. NVIDIA GPUs or AMD GPUs). We validated the correctness of our framework by comparing the results of OpenACC benchmarks against

Listing 5 Accelerator code of translation of fcw Reduction to CUDA.

```
__global__ void __generated_kernel_region_0(int* a,int min,int N,int P,int* b){
 int __kernel_getuid_x=threadIdx.x+blockIdx.x*blockDim.x;
 int i; int y;

 /* declare the shared memory of a */
 __shared__ int __kernel_smc_var_data_a[256+0+0];
 int __kernel_smc_startpointer_a;
 int __kernel_smc_endpointer_a;
 i=0+(__kernel_getuid_x);
 if(i<P){
  int tileid = (i >> TILESIZEB);
  int offsetid = (i & (TILESIZE - 1));
  { // fcw fetch begins
   __kernel_smc_startpointer_a=i-0-threadIdx.x;
   int __ipmacc_stride=blockDim.x;
   bool lastcol= blockIdx.x==(gridDim.x-1);
   __kernel_smc_endpointer_a=(lastcol)?P-1:blockDim.x+__kernel_smc_startpointer_a-1;
   int __ipmacc_length=__kernel_smc_endpointer_a-__kernel_smc_startpointer_a+1;
   int kk=0;
   kk=threadIdx.x;
   int idx=__kernel_smc_startpointer_a+kk;
   if(idx<(P) && idx>=(0)){
    __kernel_smc_var_data_a[kk]=a[idx];
   }
   __syncthreads();
  } // end of fcw fetch

  // 2 unique indexes
  // [0] i
  int __ipmacc_smc_index_a_0_dim1 = i-__kernel_smc_startpointer_a;
  // [1] i+y
  int __ipmacc_smc_index_a_1_dim1 = i+y-__kernel_smc_startpointer_a;
  { // fcw region
   for(y = min; y > 0; y >>= 1){
    if ((offsetid < y) && ((i + y) < P)) {
     int sum = __kernel_smc_var_data_a[__ipmacc_smc_index_a_0_dim1] /* replacing a [i]*/
             + __kernel_smc_var_data_a[__ipmacc_smc_index_a_1_dim1] /* replacing a [i + y]*/ ;
     __syncthreads();
     __kernel_smc_var_data_a[__ipmacc_smc_index_a_0_dim1]= sum;
     __syncthreads();
    }
   }
   if (i == (tileid * TILESIZE)){
    b[tileid] = __kernel_smc_var_data_a[__ipmacc_smc_index_a_0_dim1] /* replacing a [i]*/ ;
   }
  } // end of fcw region
 }
}
```

the serial and CUDA version. For the OpenACC version of N-Body and Matrix-matrix Multiplication, which we later compare them to FETCH_ONLY fcw, we evaluated Kepler's read-only cache by passing *const __restrict__* qualifiers. We observed less than 2% change in the measured kernel time. Hence, we report numbers assuming no read-only caches in all benchmarks.

Performance evaluations. We compile the OpenACC version of benchmarks by our framework and run it over CUDA runtime. We compare these to CUDA implementations available in CUDA SDK and Rodinia. In order to evaluate performance, we report the total time of kernel execution, kernel launch, and memory transfer between host and accelerator. We measure these times by injecting *gettimeofday()* procedure calls before and after the corresponding code. For memory transfer, we measure the time of all *cudaMemcpy* calls. We assure to use the blocking version of *cudaMemcpy* to measure the transfer time correctly. For kernel launch time, we measure the time to call kernels. For kernel execution time, we measure the time to call *cudaDeviceSynchronize()* injected immediately after every CUDA kernel launch. Measuring these numbers in OpenACC also involves the overhead of calling OpenACC API calls for mapping host memory pointers to accelerator memory pointers. The overhead involves memory transfer and kernel launch times, but not kernel execution time. Every reported number is the harmonic mean of 100 independent runs[2].

Platforms. We perform the evaluations under a CUDA-capable accelerator. We use NVIDIA Tesla K20c as the accelerator. This system uses NVIDIA CUDA 6.0 [NVIDIA Corp. 2015a] as the CUDA implementation backend. The other specifications of this system are as follows: CPU: Intel Xeon E5-2620 CPU, RAM: 16 GB, and operating system: Scientific Linux release 6.5 (Carbon) x86_64. We use GNU GCC 4.4.7 for compiling C/C++ files.

7. EXPERIMENTAL RESULTS

In this Section, we first investigate performance under fcw directive compared to baseline OpenACC and CUDA. Then we compare the development effort of these implementations. This is followed by comparing fcw to cache directive. Finally, we investigate fcw's performance sensitivity to thread block size changes.

7.1 Performance

In this section, we compare the execution time of OpenACC, fcw, and CUDA. We limit the study to the benchmarks that utilize software-managed cache. We do not report memory transfer time as it is not affected by fcw.

7.1.1 *Hotspot.* Figure 2 compares three implementations of Hotspot under different problem sizes, ranging from 128x128 to 4Kx4K. We run Hotspot for 12 iterations. CUDA and fcw implementations use the same algorithm referred to as *ghost zone optimization* [Meng and Skadron 2009]. This algorithm exploits SMC to reduce the number of global synchronization (or kernel launches). In ghost zone optimization, thread blocks *recalculate* their input data, instead of performing *global synchronization* and *reading* the output of other thread blocks. Ghost zone optimizations create halo regions around the tile assigned to each thread block. Halo regions have overlap among neighbor thread blocks and calculations for the points located in halo regions is redundant across neighbor thread blocks. Hence, the size of halo region (or amount of local iterations) makes a trade-off between global communications

[2]We also compared the measured numbers against *nvprof* [NVIDIA Corp. 2015c] and found the numbers very comparable. We found nvprof usually reports larger kernel launch and execution times, which is explained by the profiling overhead that nvprof causes. For kernel execution and memory transfer time, we measured the time that *nvprof* reports after kernels/transfers completion. For kernel launch time, we assume the time measured by *nvprof* in calling *cudaLaunch*, *cudaSetupArgument*, and *cudaConfigureCall* API procedures.

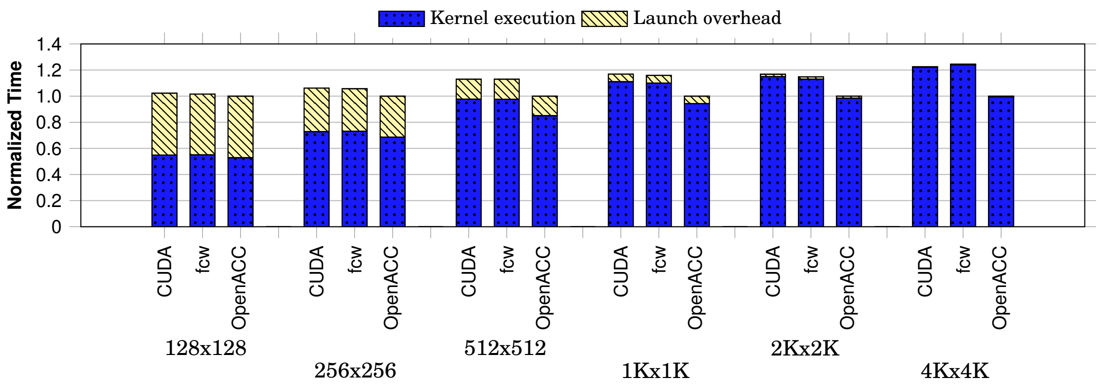
and redundant calculations [Meng and Skadron 2009]. Figure 2a and 2b report the performance under two halo region sizes (or the number of local iterations). Below we first discuss the impact of halo region size on performance. Then we compare the launch and kernel time of fcw to other implementations, followed by the discussion on the Hotspot's chip size.

halo region size. Larger halo region size impacts the performance of ghost zone implementations in two ways. On the positive side, larger halo region size decreases the amount of inter- thread block (or global) communications, reducing kernel launches and global memory accesses. On the negative side, larger halo region size increases the amount of redundant computations. Figure 2 reports the numbers under two halo region sizes, one element and two elements. Comparing fcw under the same problem sizes along Figure 2a and 2b shows the impact of halo region size on the overall time, which includes kernel launch time, global synchronization, SMC access overhead, and redundant computations. Below we explain this further.

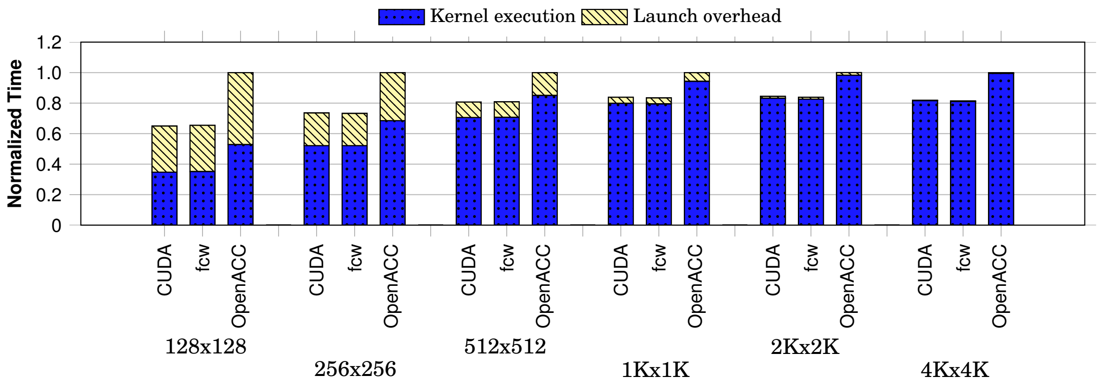
launch time. For larger problem sizes, e.g. 1K to 4K, kernel launch overhead has a insignificant share of total time. For smaller problem sizes, e.g. 128x128 and 256x256, kernel launch overhead of Hotspot has a significant share in total time. Baseline OpenACC launches several kernels to perform global synchronization across thread blocks. fcw and CUDA save significant kernel launch time by lowering global communications, by a factor proportional to halo region size (or local iterations). This reduction can be seen in 2b by comparing fcw and CUDA to OpenACC. In this case, fcw and CUDA perform two local iterations and lower the launch time nearly by 2X, compared to OpenACC.

kernel time. Under one local iteration, fcw and CUDA perform slower than OpenACC. This suggests that the overhead of redundant operations dominates the global communication savings. Additionally, fcw and CUDA execute a higher number of dynamic instructions (for accessing the data at SMC), while not taking advantage of SMC for reducing the global communication or reusing the data (since there is only one local iteration). We report one iteration to show the overheads of SMC. Generally, the number of local iterations must be larger than one to allow ghost zone to return performance improvement. Under two local iterations, fcw and CUDA start to perform faster than OpenACC. This suggests that the global communication savings is taking over redundant operations, leading to performance improvement. For the number of local iterations beyond two iterations, we found that redundant computations negate the global communication savings and always degrade performance. 2D halo regions, as used in Hotspot, have higher overhead than 1D regions since they pad the tile along two dimensions, increasing redundant computations 2X more than 1D halo regions. As we report for Pathfinder benchmark in 7.1.2, 1D halo regions allow higher improvements under larger halo regions.

problem size. fcw performs significantly better than baseline OpenACC for chip sizes smaller than 256x256. We explain this by separating the discussion of launch and kernel time. On the kernel launch time side, fcw has a huge saving in time which comes from the ghost zone optimization that performs fewer kernel launches. On the kernel time side, the kernel has a total of 16 thread blocks (chip size divided by thread block size which is 16x16) and the available thread-level parallelism is not enough to hide the memory latency effectively. Therefore, it is vital to address the global memory latency using a technique other than multi-threading. fcw takes advantage of SMC to mitigate average memory latency and improves performance. However, when the chip size increases, e.g. in 512x512, thread-level parallelism grows and provides enough threads to allow hardware to hide memory latency effectively. In these cases, immense thread-level parallelism mitigates the performance advantages of SMC in lowering average memory access latency and mitigating the gap between fcw and OpenACC. Putting together the savings of kernel and launch, performance advantages of fcw varies under different problem sizes, ranging from 23% (mostly from kernel execution improvement) in 4Kx4K chip size to 53% (equally from the kernel launch and execution improvement) in 128x128 chip size.



(a) one local iteration for CUDA and fcw.



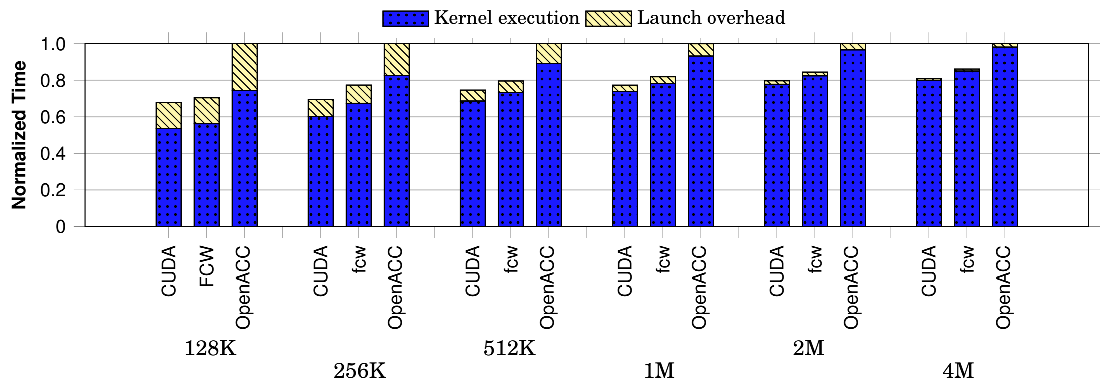
(b) two local iterations for CUDA and fcw.

Fig. 2. Kernel execution/launch time of three Hotspot implementations under different problem sizes, ranging from 128x128 to 4kx4k chip sizes. Halo region size of (a) one element and (b) two elements.

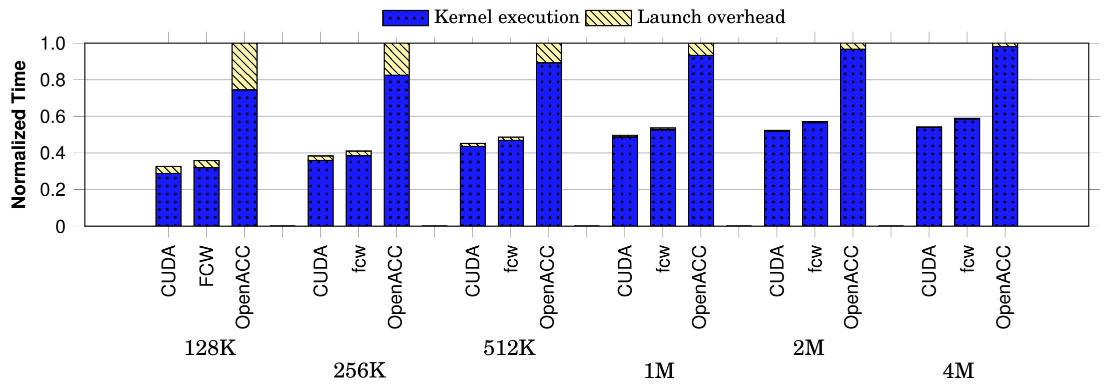
7.1.2 *Pathfinder*. Figure 3 compares performance of three Pathfinder implementations. We run the algorithm for a total of 120 iterations. fcw and CUDA use software-managed cache and implement ghost zone optimization. Ghost zone optimizations create halo regions around the tile allowing thread blocks to calculate and reuse intermediate results locally and without global communication. Each bar group reports for a particular problem size for Pathfinder (128K to 4M). Figure 3a and 3b report the numbers for different halo region sizes (different number of local iterations). Under all configurations, the CUDA implementation performs the fastest and OpenACC performs the slowest, in both kernel execution and kernel launch.

launch. Comparing kernel launch time, fcw performs close to the CUDA version and substantially faster than baseline OpenACC (80% to 6.6X). This is because CUDA and fcw implementations launch 2X lower number of kernels than OpenACC in Figure 3a (12 times lower than OpenACC in Figure 3b).

kernel. Comparing kernel execution time, fcw performs 15% to 2.3X faster than baseline OpenACC. In fcw, intermediate results are read back by threads of the same thread block, avoiding high number of global memory reads and writes, compared to OpenACC. Comparing CUDA and fcw, fcw performs 5% to 12% slower than the CUDA implementation. For each problem size, the gap between CUDA and fcw kernel execution time narrows down as the number of local iterations grows (comparing the



(a) two local iterations for CUDA and fcw.



(b) 12 local iterations for CUDA and fcw.

Fig. 3. Kernel execution/launch time of three Pathfinder implementations under different problem sizes, ranging from 128K to 4M elements. Halo size of (a) two and (b) 12 elements

same bar group in Figures 3a and 3b). This is due to the fact that our automatically generated SMC code imposes higher overhead than the hand-developed CUDA version. Increasing the halo region size (or the number of local iterations) results the computation body to dominating the initial SMC fetch overhead. Hence, the higher number of local iterations the lower the performance gap between CUDA and OpenACC. We found that after increasing the number of local iterations beyond 12, performance improvement saturates. Beyond that point, the redundant calculations of halo regions outweigh the performance advantages of lowering the number of global memory read/writes.

7.1.3 *Dyadic Convolution.* Figure 4 compares performance of OpenACC, CUDA, and fcw under various problem sizes, 2M to 64M elements per input sequence. As can be seen, CUDA and fcw perform comparable while OpenACC performs the slowest.

launch. Contribution of kernel launch time in total time is negligible (nearly 6% in 2M data).

kernel. Comparing kernel execution time, fcw performs 47% to 96% faster than baseline OpenACC. While baseline OpenACC computes the dyadic convolution directly, CUDA and fcw use Fast Walsch-Hadamard Transformation (FWHT). fcw and CUDA launch several kernels for transforming the inputs into FWHT domain, calculating dyadic convolution in FWHT domain, and transforming the output

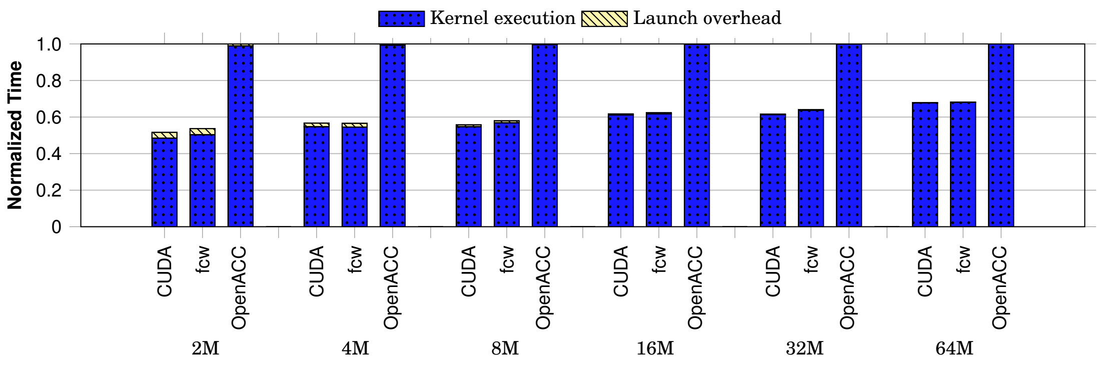


Fig. 4. Kernel execution and launch time of three Dyadic Convolution implementations: CUDA, standard OpenACC, and fcw. The legend below each group denotes the size of input sequence.

back to original domain. Generally, using FWHT algorithm minimizes irregular memory accesses of dyadic convolution and improves performance.

problem size. As the problem size grows, performance advantages of fcw and CUDA decrease. We explain this by the algorithmic difference between fcw/CUDA and OpenACC. Calculation of dyadic convolution in FWHT domain is fairly fast, but transformations of input sequences is the key bottleneck. CUDA/fcw call FWHT three times: twice to transform two input sequences and once to transform back the output sequence. We investigated the breakdown of kernel time and found that the transformations take 94% of the time. By increasing the problem size, this number grows to 96%. Transformation is performed in two kernels and only one of the kernels is optimized to use software-managed cache. The kernel which is optimized by software-managed cache accounts for 26% of transformation time. Increasing the problem size to 64M lowers this number to 20%, emphasizing that the other kernel is the bottleneck under larger problem sizes. Therefore, we can conclude that the performance advantages of CUDA/fcw is lowered under larger problem sizes due to the overhead of FWHT transformation algorithm. Further algorithmic optimization is out of the scope of this paper, as we are interested in implementing the same CUDA algorithm using fcw clause in OpenACC.

7.1.4 *N-Body*. Figure 5 compares OpenACC, fcw, and CUDA implementations of N-Body under various input sizes.

launch. CUDA performs faster in kernel launch time since OpenACC and fcw launch two kernels. OpenACC implementations calculate the summation of velocity in one kernel and the new position in another kernel. As we increase the number of bodies from 32K to 1M, kernel execution time dominates the runtime breakdown and kernel launch time becomes negligible.

kernel. Comparing kernel execution time, fcw outperforms OpenACC by 38-43%, under different problem sizes. CUDA implementation performs faster than OpenACC and fcw. CUDA outperforms OpenACC by 45-50% and fcw by 1-7%.

problem size. As shown in Figure 5, increasing the problem size slightly lowers the performance advantages of fcw over OpenACC implementation. We explain this by the difference between occupancies [3]. In the particular GPU used in this study, OpenACC, CUDA, and fcw kernels run with the occupancy of 75%, 75%, and 38%, respectively. Lower occupancy of fcw becomes a performance degrading issue under larger problem sizes where there are higher number of thread blocks. In this case, fcw

[3]Occupancy is the ratio of the number of concurrent active thread blocks to the maximum number of supported thread blocks per core.

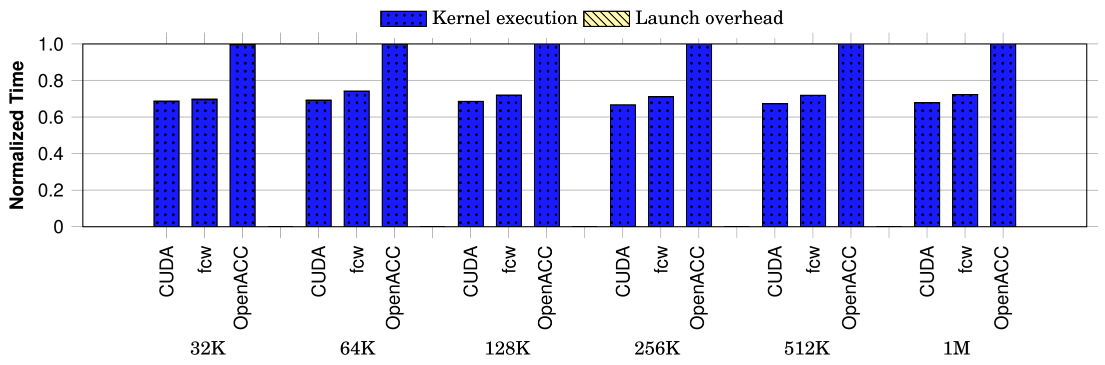


Fig. 5. Kernel execution and launch time of three N-Body implementations: CUDA, standard OpenACC, and fcw. The legend below each group denotes the number of bodies.

runs with a lower occupancy compared to the CUDA version since it overestimates the size of required software-managed cache.

7.1.5 *Matrix Multiplication.* Figure 6 compares the performance of three implementations of matrix multiplication under various input sizes.

launch. Kernel launch time is comparable under different implementations. Also increasing the problem size mitigates the contribution of kernel launch time to total runtime (less than 1%).

kernel. For kernel execution time, same behavior can be seen under different input sizes. CUDA implementation is the fastest and OpenACC implementation is the slowest. fcw outperforms OpenACC by 2.5X. CUDA implementation outperforms fcw and OpenACC by 61% and 4X, respectively. fcw and CUDA significantly reduce the global memory traffic, compared to OpenACC. Fetching tiles of input matrices in to SMC, fcw and CUDA implementations maximize memory access coalescing. Also these implementations exploit the locality among neighbor threads to minimize redundant memory fetches. Using *nvprof*, we found that fcw tiling reduces DRAM access rate by 3.94X (under 1024x1024 matrices), compared to OpenACC.

Despite the significant improvement of fcw over OpenACC, comparing CUDA and fcw, we still observe a runtime gap (61%). This gap is due to the overhead of our current implementation of fcw. Particularly, CUDA implementation shares a control-flow statement between both tiles of input matrices for fetching the data. However, in fcw implementation, two separate control-flow statements are generated for fetching each of input tiles. In our implementation of fcw, we have few heuristics to identify the opportunities for sharing the control-flow statement of SMC initialization among multiple fcw clauses. For instance, we share a fetch control-flow statement among multiple fcw clauses, if the clauses have the equal value on BEFORE and AFTER in range identifiers. However, in this matrix multiplication example, our heuristics are not capable to detect the opportunity for sharing the control-flow statement. We leave the study for finding more advanced heuristics to future work.

7.1.6 *Needleman-Wunsch.* We investigated the opportunity to implement the CUDA algorithm through fcw under Needleman-Wunsch. CUDA algorithm uses one dimensional thread blocks to fetch and writeback a two-dimensional array. Additionally, there is a complex relation between thread identifiers, global memory locations, and shared memory locations. These issues prevented us from implementing the CUDA algorithm with the simple notation of *fcw*. One possible solution maybe to add a dummy parallel loop in OpenACC to create two-dimensional thread blocks and simplify the indexing. However, this solution has two drawbacks. Firstly, it sophisticates OpenACC programming. Secondly,

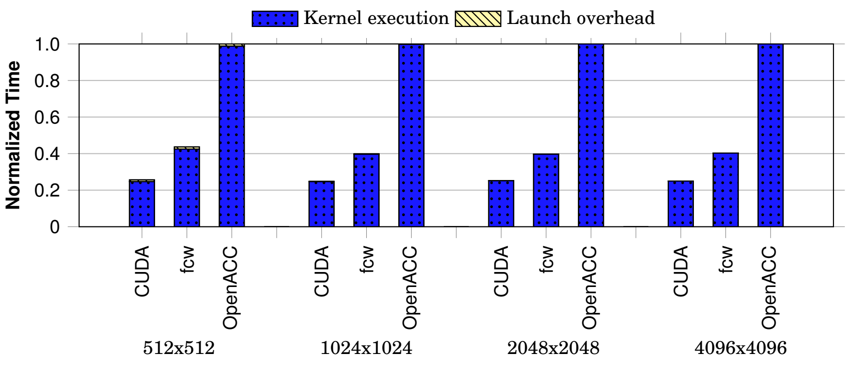


Fig. 6. Performance of OpenACC, fcw, and CUDA implementations of matrix multiplication. Each thread in OpenACC version calculates one element in the output and fetches an entire row and column from global memory. fcw and CUDA compute by fetching rows and columns in tiles into shared memory.

Table II. Comparing development effort of baseline OpenACC, fcw, and CUDA implementations in term of number of code lines.

benchmark	baseline	fcw	CUDA
Pathfinder	18	41	52
Matrix-matrix multiplication	13	23	30
Hotspot	16	42	66
N-Body	15	20	26
Dyadic Convolution	17	132	99

adding a dummy parallel loop increases the number of thread blocks remarkably and may ruin the performance advantages of SMC.

7.2 Development Effort

Table II compares development effort under fcw compared to baseline OpenACC and CUDA, in terms of number of lines of code. Since code modifications for implementing fcw is limited to the kernels region space of code, we only report the number of lines of code for this code region (not whole the code). Under CUDA, we report the number of code lines implementing the same operation. Changes are less in Pathfinder, Matrix-matrix multiplication, Hotspot, and N-Body and very high in Dyadic Convolution. The overhead in the first group is low as the modification is limited to realize tiling a loop. Tiling a loop refers to decomposing an iterative loop into two inner and outer loops and applying fcw directive over the inner loop. The overhead in the Dyadic Convolution is significant as major algorithmic changes is required. In this benchmark, fcw uses fast Walsh-Hadamard transformation which replaces the single kernel region with three kernels regions, significantly increasing code modification. Among these three kernels, only one of them exploits software-managed cache and most of the code is CUDA-like. Implementing the algorithm in fcw demands higher effort than CUDA. We found that fcw does not have significant advantage in amortizing the SMC fetch instruction, only saving two lines of code. Besides this, fcw performs roughly the same amount of instructions as the CUDA version. fcw code is larger than CUDA as there are single-line directives and loops to hint the compiler for launching parallel threads. We conclude that using fcw cannot be justified in terms of development effort in this case. Appendix A explains how we extend Pathfinder, Matrix-matrix multiplication, Hotspot, and N-Body to implement the fcw version.

7.3 Comparing to cache Directive

In this section, we report and compare performance of fcw directive to OpenACC cache directive. Similar to fcw directive, cache directive guides the compiler to cache a memory region on the chip. The memory region might be private to each thread or shared among the threads of the same thread block. In contrast to fcw directive, cache directive applies to the existing code transparently. Accordingly, the semantic of the program is not affected by considering/ignoring the directive. This difference results in significant advantage from the programming point of a view. Unfortunately it also comes with higher code overhead generated by the compiler.

With the cache directive, the compiler generates a code to check the memory addresses dynamically to decide if the address falls within the cached range or not. If the address falls within the cache range, the data is fetched from the cache, otherwise, the data is fetched from global memory. Under the fcw directive, the programmer assures that dynamic memory accesses do not violate the range specified in the fcw directive (if violated, the program may generate incorrect output.) Below we compare fcw and cache directives from the performance point of view.

cache directive is applicable to the benchmarks which exploit software-managed cache as a FETCH-_ONLY memory. Hence, we limit fcw and cache comparison to FETCH_ONLY benchmarks; matrix-matrix multiplication and N-Body simulation. For cache directive implementation of the benchmark, we use a code very similar to fcw version. The only difference is that instead of fcw directive, cache directive is used. We use our framework for compiling the cache directive, as an OpenACC implementation implementing cache directive is not available.[4]

Figure 7 and 8 report performance for fcw and cache directives under matrix-matrix multiplication and N-Body simulation, respectively. Under matrix-matrix multiplication, cache version performs 90-97% faster than the baseline OpenACC (without using software-managed cache). However, cache directive is 44-59% slower than the fcw directive. This overhead is imposed by the extra logical and control-flow instructions. This overhead is sever in N-Body. As shown in Figure 8, cache version performs always slower than the baseline OpenACC. In our experiments we make sure that the cache version and fcw version read all the data from the software-managed cache and global memory access control-flow of the cache directive is not active in the runtime. We found that the control-flow instructions in the cache version change the register usage of the kernel and nvcc compiler is unable to generate the code optimized to the degree of fcw version. We believe further investigation on this topic is needed to optimize cache directive implementation. This explains why several OpenACC implementations have limited cache directive support [Vanderbruggen et al. 2015], [Reyes et al. 2012], [Tabuchi et al. 2014].

7.4 Sensitivity to Vector Size

In this Section, we investigate the sensitivity of our findings to variations in the vector size clause of the loop directive. Since we execute OpenACC applications over CUDA, we use CUDA terminology to explain the performance impact.

Variations in the vector size impacts the number of communicating loop iterations. This in effect changes the i) number of threads in a thread block and ii) amount of CUDA shared memory used by each thread block. Since each GPU core has a certain capacity in concurrent threads and shared memory per thread block, vector size variations change the number of concurrent thread blocks per GPU core. This is often referred to as *occupancy*.

[4]We compiled the kernels with PGI Accelerator compiler 15.0 and we found out that the compiler ignores the cache directive and generates none SMC code. We also investigated few open source frameworks, e.g. RoseACC [Vanderbruggen et al. 2015], accULL [Reyes et al. 2012], and Omni compiler [Tabuchi et al. 2014], and none of them had an implementation for cache directive.

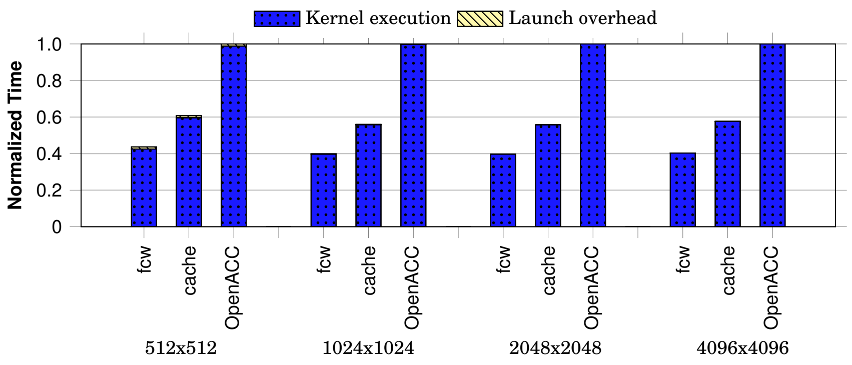


Fig. 7. Comparing performance of fcw and cache directive in matrix-matrix multiplication, under different problem sizes (512x512 to 4Kx4K matrices). Numbers are normalized to baseline OpenACC.

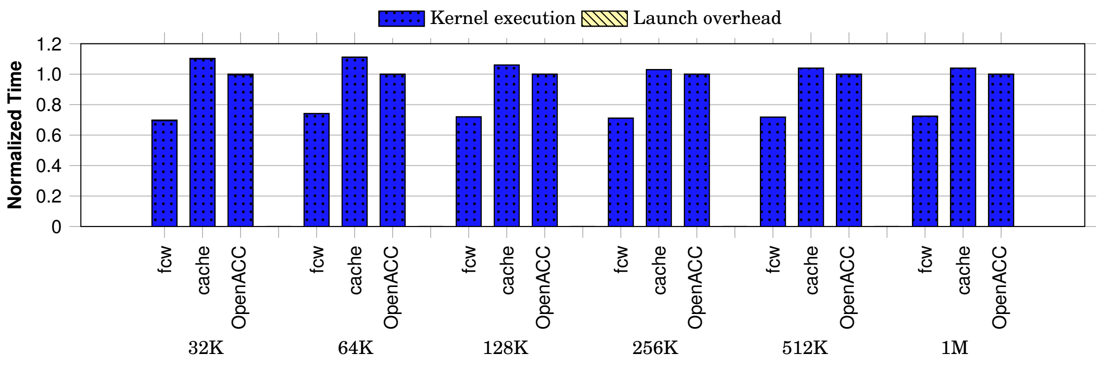
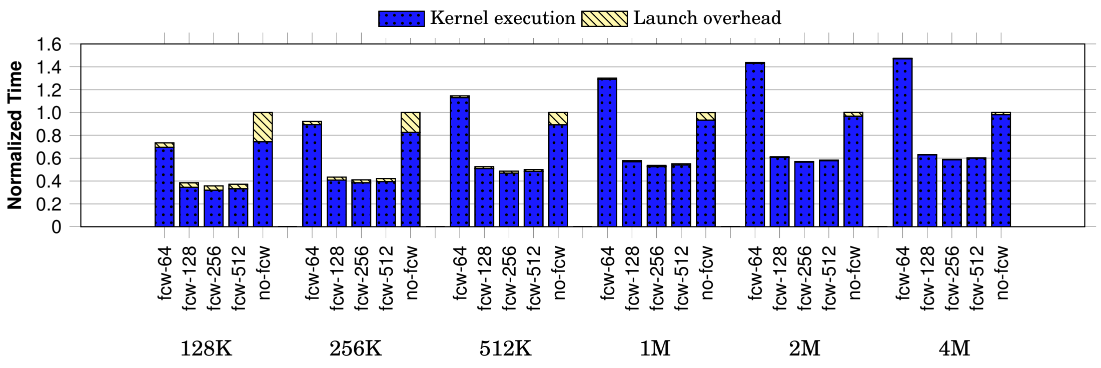


Fig. 8. Comparing performance of fcw and cache directive in N-Body simulation, under different problem sizes (32K to 1M elements). Numbers are normalized to baseline OpenACC.

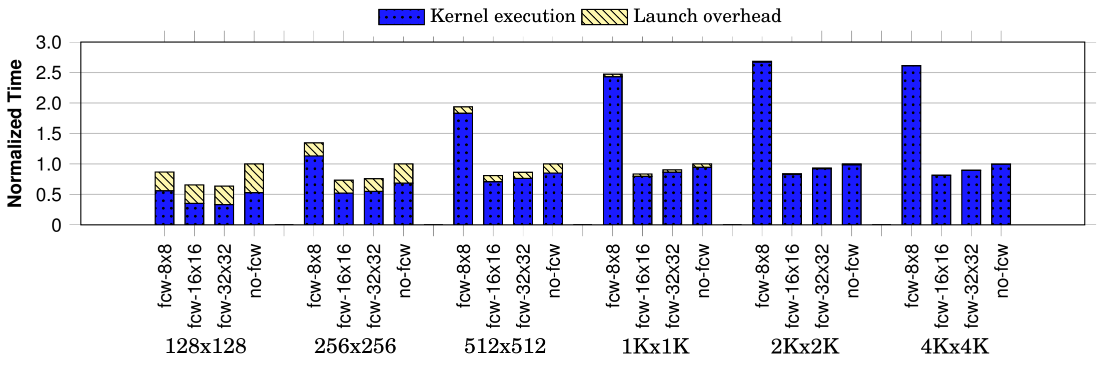
In Section 7.1.2 and 7.1.1, we investigated the performance of fcw version of Pathfinder and Hotspot benchmarks under 256 and 16x16 thread block (or vector) sizes (notice that vector size in Hotspot is specified by two numbers as Hotspot has two nested loops). Figure 9 presents performance under different thread block sizes. We evaluated occupancy for all the configurations presented in this figure. Using CUDA Occupancy Calculator [NVIDIA Corp. 2015b], the occupancy of all configurations are 100%, except Pathfinder with 64 vector size and Hotspot with 32x32 vector size which are 38% and 50%, respectively (the occupancy of baseline OpenACC is also 100% in both cases).

In both benchmarks, very small vector sizes (e.g. 64 in Pathfinder and 8x8 in Hotspot) perform poor. This is explained by the ghost zone optimization characteristic which is exploited in both Pathfinder and Hotspot. For a fixed ghost zone size (which is 12 in Pathfinder and two in Hotspot), small vector sizes increase the overhead of ghost zone operations. In these cases, the overhead of redundant calculations negates the performance gain of software-managed cache significantly leading to very poor performance.

Increasing vector size improves performance, as long as occupancy remains intact. This improvement is caused by the higher efficiency in ghost zone calculation. For example in Pathfinder, the 256 and 512 vector sizes perform best. We see slight performance degradation moving from 256 vector size to 512. This is explained by the higher overhead in thread block synchronization under larger thread blocks.



(a) Pathfinder.



(b) Hotspot.

Fig. 9. Comparing baseline OpenACC (shown with no-fcw label) and fcw, under different vector sizes and problem sizes. (a) Pathfinder: 12 local iterations and 64, 128, 256, and 512 vector sizes. (b) Hotspot: two local iterations and 8x8, 16x16, and 32x32 vector sizes.

This degradation is more sensible in Hotspot, moving from 16x16 vector size to 32x32. In this case, in addition to the synchronization overhead, 32x32 has lower occupancy which can be a critical issue under larger problem sizes.

8. DISCUSSION

8.1 Programmer or automatic compiler passes

An important question is whether achieving fcw gains through using a new directive is the most effective approach. In other words is it possible, for example, to achieve similar benefits by employing an alternative compiler. Our proposed fcw requires programmers to provide information for i) base array pointer, ii) fcw type, and iii) the data range that every loop iteration accesses. For the compiler to provide the same benefits, it has to obtain this information automatically. Instead of the first parameter, array pointer, the compiler can blindly search all arrays, which are called within the accelerator region, and investigate the potential for data caching in SMC. The second parameter, fcw type, cannot be decided properly by the compiler. Under fcw types used here, fcw can make inter-iteration communication possible. Since the compiler is blind to the algorithm, such communications may harm correctness. In this situation, the compiler is limited only to assume a safe conservative yet suboptimal FETCH_ONLY type. As for the data range, static compiler passes may be able to determine this information. This is the case when the data range identifiers do not depend on any variable with a runtime-dynamic value. We conclude that under limited circumstances only FETCH_ONLY fcw can be performed by the compiler, without any information from the programmer. CHANNEL_ONLY, FETCH_CHANNEL, CHANNEL_WB, and FETCH_CHANNEL_WB types allow fast inter-iteration communication, which needs the programmer insight and is unlikely to be achieved using compiler passes.

8.2 Applicability

fcw is applicable in two ways: implicit or explicit. In implicit mode, programmer relies on FETCH_* (fetch) and *_WB (writeback) operations of fcw and allows the compiler to implement fetch and write-back in the most efficient way. In explicit mode, programmer passes an array (it might be even a dummy array) in fcw notation (potentially CHANNEL_ONLY type) and relies on the compiler to share intermediate writes among neighbor iterations via software-managed cache. In this case, programmer explicitly fetches the data in to SMC or writes the content of SMC back to global memory, in an arbitrary user-defined pattern. The implicit mode is advantageous in term of development effort. The explicit mode is advantageous in terms of applicability. In our evaluations, we have examples of both cases. We used the implicit mode in Hotspot, Pathfinder, N-Body, and Matrix-matrix Multiplication and the explicit mode in Dyadic Convolution benchmark. We are more interested in implicit mode, since the abstraction and development effort of this mode matches the requirements of OpenACC model. Below we discuss limitations of implicit mode.

We use fcw within parallel work-sharing loops. The data range identifiers of fcw are either constant or varying, in respect to the induction variables of outer containing parallel loops. If the identifiers are constant in respect to induction variables, all parallel iterations need the same range of array elements, fully overlapped between threads. In this case, fcw is applicable if the overlapped range of data fits in the accelerator SMC. Otherwise, if the identifiers are varying in respect to induction variables, parallel iterations potentially demand different array elements. To create a communication channel among the neighbor parallel iterations, i) neighbor iterations must be mapped to the threads of the same thread block and ii) specified fcw range must have an overlap among the threads of the same thread block. Otherwise, neighbor iterations will not be able to communicate, reading intermediate writes of each other. In summary, in order to have communication between neighbor iterations under fcw the following conditions should be met: i) the programmer must perform communication only among neighbor loop iterations, ii) the compiler must map neighbor iterations to threads of the same thread block, iii) the union of data across neighbor iterations should be small enough to fit in the accelerator SMC. If necessary, the number of communicating loop iterations (or the thread block size) can be adjusted to shrink the data size and fit the data in SMC. A similar discussion applies to multi-dimensional arrays marked as fcw.

8.3 Implications

Communication thread block size. The programmer may need to control the communicating thread block size or the number of iterations that share one common SMC. She can control this number through the OpenACC vector clause of the outer loop directive. Upon translating OpenACC to CUDA (or OpenCL), the *vector* clause determines the thread block size. In nested parallel loops, a *vector* clause per *loop* can be passed to specify the thread block size along each dimension.

Data hazard in SMC and DRAM. Using fcw, the programmer may develop a code exhibiting SMC write-after-write or write-after-read data hazards. We add the following restriction to fcw definition to protect fcw under such hazards. Using implicit or explicit synchronizations, every write to a

CHANNEL_ONLY, FETCH_CHANNEL, CHANNEL_WB, and FETCH_CHANNEL_WB fcw location is guaranteed to be synchronized across the iterations falling within the same communicating thread block. Multiple writes to the same location are allowed and fcw guarantees to store one write. But the order of writes is not determined (since the threads may execute in an unspecified order on the accelerator). Similarly, there is a possibility for race conditions on global memory. This happens if the range identifiers have an overlap across different communicating thread blocks for CHANNEL_WB and FETCH_CHANNEL_WB. We also add the following restriction to fcw definition to protect fcw in such cases. When there is an overlap between different communicating thread blocks over CHANNEL_WB and FETCH_CHANNEL_WB, one memory write is guaranteed to be stored in global memory. But the order of writes is not determined.

8.4 Pointer Aliasing

Our implementation of fcw parses the fcw region and replaces every access to the specified array with an access to the cached copy in SMC. In case of the none-FETCH_ONLY fcw type, the correctness of the program might be compromised, if the memory address of the array overlaps with other arrays. To prevent such errors, we limit the use of fcw directive to the pointers which are given the property of restrict. Such pointers are declared in C with *restrict* keyword (e.g. *float *restrict ptr*). This assures that the array does not have an overlap.

9. RELATED WORK

OpenACC provides *tile* clause and *cache* directive to hint the compiler on using software-managed caches. In this section we discuss the capabilities and limitations of these clauses in exploiting them as shared memory of CUDA. *tile* is a clause used by loop directive to split the loop into two loops; outer and inner. This clause is intended to specify a boundary for splitting the loop calculations over different gangs. Programmers do not contribute to tiling adjustments as the compiler carries the necessary tiling optimizations. Although this clause introduces the tiling concept of CUDA/OpenCL in OpenACC programming model, there are still two limitations in employing it for utilizing SMC effectively. First, unlike CUDA/OpenCL, *tile* does not provide enough information to allow the compiler exploit the software-managed cache as an inter-thread communication medium. Second, developers cannot control the software-managed cache at the programming level via this clause. *tile* directive improves the locality of accesses and assists the accelerator in fitting the working set of gangs in the small-sized automatic hardware cache. *cache* directive can be used within work-sharing loops to fetch a memory region into the highest level of accelerator's cache hierarchy. It can mark a subset of array elements to exploit locality efficiently. Although the compiler can use this hint to reduce costly global memory accesses, there are opportunities that cannot be captured by *cache*. For example, as we have shown in this study, communication between neighbor iterations of a work-sharing loop, similar to the communication across the threads within a CUDA thread block, is not possible under *cache*. Sharing intermediate memory writes among neighbor iterations can reduce global memory communications significantly. In summary, *tile* clause and *cache* directive do not allow communication through SMC and they can be considered as a specific type of fcw, FETCH_ONLY. A previous work [Hoshino et al. 2013] also found the same limitation in the *cache* directive.

Lee and Vetter [Lee and Vetter 2014] introduce a framework for compiling, debugging, and profiling OpenACC applications. They also introduce openarc directive to allow OpenACC programmer to map OpenACC arrays to different CUDA memory spaces, including shared and texture memory spaces. They do not investigate the effectiveness of their proposal for these mappings. Based on the short introduction that they present, we believe their proposal for utilizing shared memory is different from ours in two ways. Firstly, while openarc directive needs programmer to separate shared memory array and corresponding global memory array in the code, fcw separates the arrays automatically, based on the information presented by the programmer. Secondly, while openarc directive allows fine-grained control to OpenACC programmer to perform fetch, synchronization, and writeback, fcw handles these operations implicitly. Based on these differences, we consider fcw as a high-level proposal for utilizing SMC and openarc as a low-level fine-grained control over SMC.

Hoshino et al. [Hoshino et al. 2014] investigate the impact of memory layout on the performance of NVIDIA Kepler, Intel XeonPhi, and Intel Xeon processors, under directive-based programming languages. They found that having structure-of-arrays is much more efficient than array-of-structures under Kepler and XeonPhi, while it has minor impact on the performance of Xeon. They explain this by relatively smaller cache of Kepler (110 Bytes per hardware thread) and XeonPhi (128 KBytes per hardware thread), compared to Xeon (1048 KBytes per hardware thread). They also introduce a new directive allowing programmer to change the data layout of multi-dimensional arrays.

Li et al. [Li et al. 2014] compare the performance of software-managed and hardware-managed caches in NVIDIA GPUs. They study several test cases and compare two CUDA implementations. The first implementation uses shared memory and the second implementation tiles computations and allows hardware-managed cache to store the tiles implicitly. They showed that software-managed cache can use shared memory to avoid conflict miss. Moreover, it has higher bandwidth, higher memory-level parallelism, and less serialization. They also reported that hardware-managed cache executes lower number of dynamic instructions and has higher occupancy.

Reyes et al. [Reyes et al. 2012] introduce an open-source tool, accULL, to execute OpenACC applications on accelerators. The tool consists of a source to source compiler and a runtime library. Tian et al. [Tian et al. 2013] introduce an OpenACC implementation integrated in OpenUH [Liao et al. 2007]. They evaluate the impact of mapping loop iterations over GPU parallel work-items. Nakao et al. [Nakao et al. 2014] introduce XACC as an alternative to MPI+OpenACC programming model to harness the processing power of cluster of accelerators. Murai et al. [Murai et al. 2014] propose an extension to OpenACC to facilitate multi-accelerator processing. Language extension includes on_device clause for data, kernels, and loop directives.

10. CONCLUSION

Recent efforts have introduced OpenACC as a promising programming model saving abundant efforts in accelerator programming. In this study, we investigated the performance gap between CUDA and OpenACC implementations of several different applications. We found that CUDA's programming flexibility in exploiting software-managed cache is the key factor widening the gap between CUDA and OpenACC. Based on this observation, we proposed a new OpenACC directive, referred to as fcw, to allow achieving benefits which are possible by using the accelerator software-managed cache through high-level OpenACC API. In conjunction with fcw, we introduce a new communication model in OpenACC allowing neighbor iterations of parallel loops to communication and share intermediate data. Using fcw, the OpenACC programmer can fetch a tile of data from DRAM to the software-managed cache, share intermediate data between iterations, and write the final value back to global memory. We studied wide set of benchmarks to show the applicability and performance advantages of fcw compared to OpenACC and CUDA. Compared to CUDA, we found programming in OpenACC allows simplifying shared memory programming for two reasons. First, the shared memory implementation can be obtained with fewer lines of code. Secondly, the compiler can automatically inject debugging and verification routines which simplifies debugging. We also discussed the applicability and performance limitations of fcw compared to CUDA. We showed that fcw can narrow down the gap between OpenACC and CUDA significantly. We presented an efficient implementation of fcw on CUDA-capable accelerators (a very similar implementation can be derived for OpenCL-capable accelerators). Assuming there are efficient CUDA or OpenCL implementations available on OpenACC accelerators, OpenACC's portability is not compromised under the proposed directive.

11. ACKNOWLEDGEMENTS

We thank anonymous reviewers for their valuable comments on this work. This work was supported by the Natural Sciences and Engineering Research Council of Canada, Discovery Grants Program.

REFERENCES

AMD INCORPORATION. 2007. AMD's close-to-the-metal. Available: http://sourceforge.net/projects/amdctm/. Accessed: 2015-06-19.

ARNDT, J. 2011. *Matters computational*. Springer, Chapter 23.

BUCK, I. 2004. BrookGPU. Available: http://graphics.stanford.edu/projects/brookgpu/. Accessed: 2015-06-19.

CHE, S., BOYER, M., MENG, J., TARJAN, D., SHEAFFER, J. W., LEE, S.-H., AND SKADRON, K. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*. IISWC '09. IEEE Computer Society, Washington, DC, USA, 44–54.

GARLAND, M., LE GRAND, S., NICKOLLS, J., ANDERSON, J., HARDWICK, J., MORTON, S., PHILLIPS, E., ZHANG, Y., AND VOLKOV, V. 2008. Parallel Computing Experiences with CUDA. *IEEE Micro 28,* 4, 13–27.

HOSHINO, T., MARUYAMA, N., AND MATSUOKA, S. 2014. An OpenACC Extension for Data Layout Transformation. In *Proceedings of the First Workshop on Accelerator Programming Using Directives*. WACCPD '14. IEEE Press, Piscataway, NJ, USA, 12–18.

HOSHINO, T., MARUYAMA, N., MATSUOKA, S., AND TAKAKI, R. 2013. CUDA vs OpenACC: Performance Case Studies with Kernel Benchmarks and a Memory-Bound CFD Application. In *Proceeding of the 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2013*. 136–143.

KAPASI, U., RIXNER, S., DALLY, W., KHAILANY, B., AHN, J., MATTSON, P., AND OWENS, J. 2003. Programmable Stream Processors. *Computer 36,* 8, 54–62.

LASHGAR, A., MAJIDI, A., AND BANIASADI, A. 2014. IPMACC: Open Source OpenACC to CUDA/OpenCL Translator. arXiv:1412.1127v1 [cs.PL].

LEE, S. AND VETTER, J. S. 2014. OpenARC: Extensible OpenACC Compiler Framework for Directive-based Accelerator Programming Study. In *Proceedings of the First Workshop on Accelerator Programming Using Directives*. WACCPD '14. IEEE Press, Piscataway, NJ, USA, 1–11.

LI, C., YANG, Y., DAI, H., YAN, S., MUELLER, F., AND ZHOU, H. 2014. Understanding the tradeoffs between software-managed vs. hardware-managed caches in GPUs. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*. 231–242.

LIAO, C., HERNANDEZ, O., CHAPMAN, B., CHEN, W., AND ZHENG, W. 2007. OpenUH: An Optimizing, Portable OpenMP Compiler: Research Articles. *Concurr. Comput. : Pract. Exper. 19,* 18, 2317–2332.

LINDHOLM, E., NICKOLLS, J., OBERMAN, S., AND MONTRYM, J. 2008. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE, Micro 28,* 2, 39–55.

MARK, W. R., GLANVILLE, R. S., AKELEY, K., AND KILGARD, M. J. 2003. Cg: A System for Programming Graphics Hardware in a C-like Language. *ACM Trans. Graph. 22,* 3, 896–907.

MENG, J. AND SKADRON, K. 2009. Performance Modeling and Automatic Ghost Zone Optimization for Iterative Stencil Loops on GPUs. In *Proceedings of the 23rd International Conference on Supercomputing*. ICS '09. ACM, New York, NY, USA, 256–265.

MURAI, H., NAKAO, M., SHIMOSAKA, T., TABUCHI, A., BOKUT, T., AND SATO, M. 2014. XcalableACC - a Directive-based Language Extension for Accelerated Parallel Computing. In *Proceedings of the Supercomputing Conference poster*. SC '14 poster session. Piscataway, NJ, USA.

NAKAO, M., MURAI, H., SHIMOSAKA, T., TABUCHI, A., HANAWA, T., KODAMA, Y., BOKUT, T., AND SATO, M. 2014. XcalableACC: Extension of XcalableMP PGAS Language Using OpenACC for Accelerator Clusters. In *Proceedings of the First Workshop on Accelerator Programming Using Directives*. WACCPD '14. IEEE Press, Piscataway, NJ, USA, 27–36.

NICKOLLS, J., BUCK, I., GARLAND, M., AND SKADRON, K. 2008. Scalable Parallel Programming with CUDA. *Queue 6,* 2, 40–53.

NVIDIA CORP. 2015a. CUDA Toolkit 6.0. Available: https://developer.nvidia.com/cuda-downloads. Accessed: 2015-06-19.

NVIDIA CORP. 2015b. NVIDIA CUDA Occupancy Calculator. Available: http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls. Accessed: 2015-06-19.

NVIDIA CORP. 2015c. Profiler's User Guide. Available: http://docs.nvidia.com/cuda/profiler-users-guide/. Accessed: 2015-06-19.

NYLAND, J. P. L. AND HARRIS, M. 2007. *Gpu Gems 3: chapter 31* First Ed. Addison-Wesley Professional.

OPENACC. 2015. The OpenACC Application Programming Interface. Available: http://www.openacc-standard.org/. Accessed: 2015-06-19.

PATHSCALE. 2013. Modified Rodinia Benchmark Suite. Available: https://github.com/pathscale/rodinia. Accessed: 2015-06-19.

REYES, R., LÓPEZ-RODRÍGUEZ, I., FUMERO, J. J., AND DE SANDE, F. 2012. accULL: An OpenACC Implementation with CUDA and OpenCL Support. In *Proceedings of the 18th International Conference on Parallel Processing*. Euro-Par'12. Springer-Verlag, Berlin, Heidelberg, 871–882.

TABUCHI, A., NAKAO, M., AND SATO, M. 2014. A Source-to-Source OpenACC Compiler for CUDA. In *Euro-Par 2013: Parallel Processing Workshops*. Lecture Notes in Computer Science Series, vol. 8374. Springer Berlin Heidelberg, 178–187.

THE KHRONOS GROUP. 2015. OpenCL: The open standard for parallel programming of heterogeneous systems. Available: https://www.khronos.org/opencl/. Accessed: 2015-06-19.

THIES, W., KARCZMAREK, M., AND AMARASINGHE, S. P. 2002. StreamIt: A Language for Streaming Applications. In *Proceedings of the 11th International Conference on Compiler Construction*. CC '02. Springer-Verlag, London, UK, UK, 179–196.

TIAN, X., XU, R., YAN, Y., YUN, Z., CHANDRASEKARAN, S., AND CHAPMAN, B. 2013. 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*. LCPC 2013.

VANDERBRUGGEN, T. ET AL. 2015. Roseacc. Available: http://roseacc.org/. Accessed: 2015-06-19.

WIENKE, S., SPRINGER, P., TERBOVEN, C., AND AN MEY, D. 2012. OpenACC: First Experiences with Real-world Applications. In *Proceedings of the 18th International Conference on Parallel Processing*. Euro-Par'12. Springer-Verlag, Berlin, Heidelberg, 859–870.

WONG, H., PAPADOPOULOU, M.-M., SADOOGHI-ALVANDI, M., AND MOSHOVOS, A. 2010. Demystifying GPU Microarchitecture Through Microbenchmarking. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*. 235–246.

A. CODE MODIFICATION

A.1 Pathfinder

Listing 6 and 7 show baseline and fcw implementations of Pathfinder. The fcw version implements the ghost zone algorithm and differs from the baseline in four ways. Firstly, the most outer loop in the baseline is divided into inner and outer loops. Secondly, the number of iterations of the parallel loop is increased to allow enough parallel threads for redundant ghost zone calculations. Also part of the code maps the loop inductive variable to the output index. Thirdly, fcw directive is added over the most inner loop. The fcw type is FETCH_CHANNEL which allows intermediate writes to be visible to neighbor threads. Finally, the final state of the local copy (in the software-managed cache) is written back by the loop iterations that were active in recent iterations. True value of the boolean variable, named *computed* in the code, indicates that the thread was active in the last inner iteration of the ghost zone and should write the final value back to global memory.

Listing 6 Pathfinder baseline OpenACC.

```
#pragma acc data copyin(src[0:cols],data[0:rows*cols]) copyout(dst[0:cols])
{
  for (int t = 0; t < rows-1; t++) {
    temp = src;
    src = dst;
    dst = temp;
    #pragma acc kernels
    #pragma acc loop private(min) independent
    for(int n = 0; n < cols; n++){
      min = src[n];
      if (n > 0)
```

```
        min = MIN(min, src[n−1]);
      if (n < cols−1)
        min = MIN(min, src[n+1]);
      dst[n] =  (data[(t+1)∗cols+n]) + min ;
    }
  }
}
```

Listing 7 Pathfinder fcw OpenACC.

```
#define TILESIZE 256
#define TILESIZELOG 8
#pragma acc data copyin(src[0:cols],data[0:rows∗cols]) copyout(dst[0:cols])
{
  for (int t=0; t < rows−1; t+=inner_iter){
    temp = src;
    src = dst;
    dst = temp;
    int pad=((cols/(TILESIZE−2∗inner_iter))+1)∗inner_iter∗2;
    #pragma acc kernels
    #pragma acc loop private(min) independent vector(TILESIZE)
    for(int np=0; np<(cols+pad); np++){
      int n=np−((np>>TILESIZELOG)∗2∗inner_iter+inner_iter);
      int localid=np&(TILESIZE−1);
      int val=0;
      bool computed=false;
      #pragma acc fcw FETCH_CHANNEL(src[n:0:0])
      {
        for(int i=0; i<inner_iter; i++){
          int row=t+i+1;
          if(row<rows){
            computed=false;
            if(localid>i && localid<(TILESIZE−i−1) && n>=0 && n<cols){
              min = src[n];
              computed=true;
              if (n > 0)
                min = MIN(min, src[n−1]);
              if (n < (cols−1))
                min = MIN(min, src[n+1]);
              val = (data[row∗cols+n]) + min ;
              src[n]=val;
            }
          }
        }
      }
      if(computed){
        dst[n]=val;
      }
    }
  }
}
```

A.2 Matrix-matrix Multiplication

Listing 8 and 9 show baseline and fcw implementations of Matrix-matrix Multiplication. The fcw implementation differs from the baseline in two ways. Firstly, the most inner loop is decomposed into inner and outer loops allowing a different tile of data to be processed each time. Secondly, the fcw directive is added over the inner loop to cache the active tile.

Listing 8 Matrix-matrix multiplication baseline OpenACC.

```
#pragma acc data pcopyin(a[0:LEN][0:LEN],b[0:LEN][0:LEN]) pcopyout(c[0:LEN][0:LEN])
#pragma acc kernels
#pragma acc loop independent
for (i = 0; i < LEN; ++i) {
  #pragma acc loop independent
  for(j=0; j<LEN; j++){
    float sum=0;
    for(m=0; m<LEN; m++){
      sum += a[i][m]*b[m][j];
    }
    c[i][j]=sum;
  }
}
```

Listing 9 Matrix-matrix multiplication fcw OpenACC.

```
#define MIN(a,b) ((a<b)?a:b)
#pragma acc data pcopyin(a[0:LEN][0:LEN],b[0:LEN][0:LEN]) pcopyout(c[0:LEN][0:LEN])
#pragma acc kernels
#pragma acc loop independent vector(16)
for(i=0; i<LEN; ++i){
  #pragma acc loop independent vector(16)
  for(j=0; j<LEN; ++j){
    float sum=0;
    for(l=0; l<LEN; l+=16){
      int offseti=(i&(16-1))+l;
      int offsetj=(j&(16-1))+l;
      #pragma acc fcw FETCH_ONLY(a[i:0:0:offsetj:0:0],b[offseti:0:0:j:0:0])
      if(j<LEN && i<LEN){
        for(m=l; m<MIN(l+16,LEN); m++){
          sum += a[i][m]*b[m][j];
        }
      }
    }
    if(j<LEN && i<LEN){
      c[i][j]=sum;
    }
  }
}
```

A.3 Hotspot

Listing 10 and 11 show baseline and fcw implementation of Hotspot. Modifications are very similar to the modifications in Pathfinder (Section A.1) as both benchmarks implement the ghost zone algorithm. Hotspot is more complex than Pathfinder, since Hotspot implements the two-dimensional ghost zone. Here fcw code differs from the baseline in five ways in implementing the two-dimensional ghost zone algorithm. Firstly, iterations of the outer row and column parallel loops are increased to supply enough threads for the redundant ghost zone calculations. Also the output identifier of each thread is calculated in early lines of each loop. Secondly, the fcw directive is added over the inner loop to cache intermediate writes. The fcw type is FETCH_CHANNEL which makes intermediate writes visible to neighbor loop iterations. Thirdly, an inner loop is added to perform local iterations. Fourthly, a guard is added over the calculation body of the inner loop to specify the threads which must be active in each iteration of the ghost zone. Lastly, the final value of the software-managed cache is written back to global memory.

Listing 10 Hotspot baseline OpenACC.

```
#pragma acc kernels present(temp, power, result)
#pragma acc loop independent
for (r = 0; r < row; r++) {
  #pragma acc loop independent
  for (c = 0; c < col; c++) {
    int S=(r==(row−1))?row−1:r+1;
    int N=(r==(0))?0:r−1;
    int W=(c==(0))?0:c−1;
    int E=(c==(col−1))?col−1:c+1;
    delta = (stepCap) ∗ (power[r][c] +
        (temp[S][c] + temp[N][c] − 2.0∗temp[r][c])∗rRy +
        (temp[r][E] + temp[r][W] − 2.0∗temp[r][c])∗rRx +
        (amb_temp − temp[r][c])∗rRz);
    result[r][c] =temp[r][c]+ delta;
  }
}
```

Listing 11 Hotspot fcw OpenACC.

```
int dimrow=(row+((2∗innerIter)∗(row/(16−2∗innerIter)+1)));
int dimcol=(col+((2∗innerIter)∗(col/(16−2∗innerIter)+1)));
#pragma acc kernels present(temp, power, result, written)
#pragma acc loop independent vector(16)
for (rs=0; rs<dimrow; rs++) {
  int r=rs−((rs>>4)∗2∗innerIter+innerIter);
  #pragma acc loop independent vector(16)
  for (cs=0; cs<dimcol; cs++) {
    int c=cs−((cs>>4)∗2∗innerIter+innerIter);
    double new_temp;
    bool compute=false;
    int S=(r==(row−1))?row−1:r+1;
    int N=(r==(0)) ?0:r−1;
    int W=(c==(0)) ?0:c−1;
    int E=(c==(col−1))?col−1:c+1;
    #pragma acc fcw FETCH_CHANNEL(temp[r:0:0:c:0:0],power[r:0:0:c:0:0])
    {
      for(int iter=0; iter<innerIter; iter++){
        compute=false;
        if( (r>=0) && (c>=0) && (r<row) && (c<col) &&
            ((( rs&(16−1))>iter) && ((cs&(16−1))>iter)) &&
            ((( rs&(16−1))<(16−iter−1))&&((cs&(16−1))<(16−iter−1))) ){
          compute=true;
          delta = (stepCap) ∗ (power[r][c] +
              (temp[S][c] + temp[N][c] − 2.0∗temp[r][c]) ∗ rRy +
              (temp[r][E] + temp[r][W] − 2.0∗temp[r][c]) ∗ rRx +
              (amb_temp − temp[r][c]) ∗ rRz);
          new_temp= delta + temp[r][c];
        }
        if(iter==(innerIter−1))
          break;
        if(compute){
          temp[r][c] = new_temp;
        }
      }
    }
    if(compute){
      result[r][c] = new_temp;
    }
  }
}
```

A.4 N-Body

N-Body simulation has two kernels where one of them can be optimized to take advantage of software-managed cache. Listing 12 and 13 show this kernel under baseline and fcw implementations. There are two differences between the baseline and fcw. Firstly, the most inner loop is decomposed to inner and outer to perform operations on tiles. Secondly, the fcw directive is added over the inner loop to cache the active tile in the software-managed cache.

Listing 12 N-Body baseline OpenACC.

```
#pragma acc data pcopyin(m_force[0:m_numBodies],m_pos[0:m_numBodies])
#pragma acc kernels
#pragma acc loop independent
for (int i = 0; i < m_numBodies; i++){
  float3 acc;
  acc.x=0;
  acc.y=0;
  acc.z=0;
  float4 pos_p=m_pos[i];
  int j = 0;
  for(j=0; j< m_numBodies; j++){
    bodyBodyInteraction(&acc, pos_p, m_pos[j], m_softeningSquared);
  }
  m_force[i] = acc;
}
```

Listing 13 N-Body fcw OpenACC.

```
#define TILESIZE 256
#define TILESIZELOG 8
#pragma acc data pcopyin(m_force[0:m_numBodies],m_pos[0:m_numBodies])
#pragma acc kernels
#pragma acc loop independent
for(int i=0; i<m_numBodies; i++){
  float3 acc;
  acc.x=0;
  acc.y=0;
  acc.z=0;
  float4 m_pos_local = m_pos[i];
  for(int tile=0; tile<=((m_numBodies>>TILESIZELOG)+1); tile++){
    int bound = bnd1<m_numBodies?((tile+1)*TILESIZE):m_numBodies;
    #pragma acc fcw FETCH_ONLY(m_pos[(tile*TILESIZE):0:TILESIZE])
    for(int j=(tile*TILESIZE); j<bound; j++){
      bodyBodyInteraction(&acc, m_pos_local, m_pos[j], m_softeningSquared);
    }
  }
  m_force[i]=acc;
}
```